# Interactive Theorem Assistant for Learning "A Logical Approach to Discrete Math": Proof Theory and Assessment Experience

**Federico Flaviani, Jorge Baralt-Torrijos, Soraya Carrasquel and David Coronado**
Universidad Simón Bolívar,
Departamento de Computación y Tecnología de la Información,
Caracas, Venezuela, 1080-A,
{*fflaviani, jbaralt, scarrasquel, dcoronado*}*@usb.ve*

**Abstract**

CalcLogic is a proof assistant adapted for students of Calculational Logic. CalcLogic uses the same syntax and theorems as in Gries and Schneider's book, "A Logical Approach to Discrete Math". All proofs can be done by students using only the mouse, without the need to learn any language. To mechanize the assistant, a new proof theory for Calculational Logic is developed in this paper. This theory allows the loading of first-order theories axiomatized with an axiom scheme. Additionally, a study is presented using statistical evidence, that suggests an improvement in learning Calculational Logic when the formative activities are made by means of the assistant.

**Keywords:** Calculational Logic, Predicate Logic, Proof Theory, Lambda Calculus, Interactive Theorem Prover, Education.

## 1 Introduction

A proof assistant is a software tool that helps mathematicians to construct proofs that can be mechanically verified by a computer. The system can either generate proofs on its own or in an interactive user-assisted manner.

Calculational Logic is defined by Dijkstra and Scholten [1]. However, this formal system was made popular by the book "A Logical Approach to Discrete Math" [2], hereafter abbreviated as LADM. CalcLogic [3] is an interactive, web-based tool designed for proving. The first published version was developed during the COVID-19 pandemic as a solution to remote assessments of discrete mathematics [4].

Since then, there has been incremental development, expanding functionalities and even re-engineering its design. Each new version of the assistant has allowed students to work with a different mathematical theory, and in each case, educational experiments were conducted to validate the degree of student acceptance. With the first version, an education experiment with finite differences, summations and field theory [4] was carried out. In the second version, an experiment with propositional logic and Boolean algebra was carried out [5, 6]. In these versions, all statements were equalities, all objects had the same type and quantification was not allowed.

This paper introduces a new version of CalcLogic that supports objects of different types, includes quantifiers, and the user can now prove statements other than equalities. With this new expressive power, it is possible to load a theory of predicate logic, based on chapters 3, 4, 8 and 9 of LADM. Since CalcLogic is web-based, students can complete the proofs from home, so they are all stored in a centralized database. This way, teachers can see each student's progress individually.

The students showed rapid adaptation to the interface and great speed in solving exercises when using the assistant. Designed for a step-by-step user experience, CalcLogic processes one user-submitted inference at a time. For each input, the system either notifies the user of any errors or executes the inference and prints its result. CalcLogic does not inform you why an inference is wrong; the users must discover the errors themselves, otherwise the application will not allow them to move forward. Each inference is printed according to the Dijkstra-Scholten notation [1, 2].

This paper is an extended version of [7], which focused on presenting educational experiments to measure user interface acceptance and its impact on learning. This version adds the explanation of a new proof

theory that supports the mechanization of the assistant. This theory, developed exclusively for this work, is published for the first time in this extended version and is completely different from that used by modern proof assistants. Even this new proof theory employs a simpler type system.

At its core, calculational logic works by replacing expressions with their equal or equivalent counterparts, just like when a calculation is written on a blackboard. In our experience, calculational proofs tend to be shorter than those used in other systems. We've also noticed that students learn these types of proof quickly. This is why we decided to implement a Calculational Logic-based assistant.

Replacement in predicate logic is not easy to define. For this reason, extending the assistant for this type of logic required a complete re-engineering of its inference core. An example of this difficulty is explained below.

In Calculational Logic, $E[z := p]$ means the substitution of $p$ for each occurrence of the variable $z$ in the formula $E$ (bound variables in $E$ are renamed if necessary). In propositional logic, converting the formula $E[z := p]$ to $E[z := q]$ can be understood as replacing $p$ by $q$ within the formula $E[z := p]$. For example, if we want to replace $\neg\neg q$ by $q$ in the formula $p \wedge (\neg\neg q \vee r)$, we can rewrite this formula as $E[z := \neg\neg q]$ where $E$ is $p \wedge (z \vee r)$. Then $E[z := q]$ is the proposition $p \wedge (q \vee r)$, which is the formula resulting from the desired replacement.

In the formula $E$, $z$ occurs only once. This variable is used to locate the position where the replacement takes place. However, defining the replacement using $E$, $[z := p]$, and $[z := q]$ is not possible in predicate logic. For example, if we want to replace $x + 0$ by $0 + x$ in $(\forall x| : x + 0 = x)$, and we use $E$ as $(\forall x| : z = x)$, then $E[z := x + 0]$ is $(\forall y| : x + 0 = y)$ and $E[z := 0 + x]$ is $(\forall y| : 0 + x = y)$. As shown, there is no way to obtain a textual replacement, since the bound variable $x$ in $\forall$ is always renamed. The operator $[z := p]$ renames the bound $x$ in $E$, whenever $x$ occurs free in $p$ and substituting it would cause a name conflict.

The textual replacement is defined as a metatheorem in Hilbert-style logics. Its name is "Replacement Metatheorem", and it is proven by making several derivations by induction over the size of the formula $E$. For example, in the formal system of natural deduction to replace $x + 0$ by $0 + x$ within the formula $(\forall x| : x+0 = x)$, the following steps must be performed using the replacement axiom $e = t \wedge E[z := e] \Rightarrow E[z := t]$:

1. $(\forall a, b)(a + b = b + a)$ (premise)
2. $(\forall x)(x + 0 = x)$ (premise)
3. $x + 0 = 0 + x$ ($\forall$ elimination on 1, with $a, b := x, 0$)
4. $x + 0 = x$ ($\forall$ elimination on 2)
5. $x + 0 = 0 + x \ \wedge \ x + 0 = x$ ($\wedge$ introduction using 3 and 4)
6. $x + 0 = 0 + x \ \wedge \ (z = x)[z := x + 0] \Rightarrow (z = x)[z := 0 + x]$ (replacement axiom with $E : z = x$)
7. $0 + x = x$ (modus ponens using 5 and 6)
8. $(\forall x)(0 + x = x)$ ($\forall$ introduction using 7 since $x$ was arbitrary)

To mechanize an assistant based on this metatheorem requires performing all these inferences every time a replacement is made. This can generate a lot of computation, which this work seeks to improve.

This work consists of two sections. The first develops a proof theory for mechanizing Calculational Logic. This theory seeks to define replacement (in the presence of predicates) in a simpler way, without using the Replacement Metatheorem and without modifying any aspect of LADM. This first section is not found in a previous version of this work [7].

In the second section an experiment is reported with students of Symbolic Logic at Simón Bolívar University. Symbolic Logic is an introductory course of Calculational Logic at the beginning of the career in Computer Engineering. The principal book of the course is LADM. The tool was used to support formative activities in the learning of Propositional Logic and Predicate Logic.

Prior to this work, another experiment was carried out [5]. It was shown that the grades the student obtained in assessments made by the tool corresponded to the grades obtained in a written assessment of the same topic. It was concluded that this was because CalcLogic is a repository of student-done exercises that was automatically checked. This repository is a certificate of the number of correct exercises that the student has done during their studies, which at the same time certifies that the student has been studying correctly.

However, due to the way the experiment was conducted, it was not possible to determine whether the students acquired the subject competencies in the standard way and then learned to use the tool. Therefore, it was not possible to conclude whether the use of the tool helped them acquire these competencies.

In this work, new functionalities of CalcLogic are presented and the experiment sought to answer if the use of the tool help to acquire competencies. We also wanted to measure how users adapt to the graphical interface.

## 2 Contribution

This paper presents a new version of CalcLogic in which the backend inference module was re-engineered. The changes were the result of the belief that Dijkstra's essential ideas for his Calculational Logic were functional at style of Church (Lambda Calculus)[8], where predicates and quantifiers are functions. We believe that some of Dijkstra's ideas can be written more clearly using the languages that Church or Curry used [8, 9].

In fact, Dijkstra emphasizes the functional structure that formulas could had [10]. For example, on page 2 he writes that if $X$ and $Y$ are predicates that depend on $s$, then the repertoire of formulas $X.s \vee Y.s$, $X.s \wedge Y.s$, $\neg X.s$, etc., can be written as $(X \vee Y).s$, $(X \wedge Y).s$, $(\neg X).s$, etc. He then says that with a rich repertoire of function identities as before, the number of functional applications of any formula $P$ can reduce to 1, and would be written as $P.s$ where $P$ was called structure (for example $(\neg X.s \wedge Y.s) \vee Z.s$ can be written as $((\neg X \wedge Y) \vee Z).s$ and the function $(\neg X \wedge Y) \vee Z$ is a structure). This statement is evidently true using the language of Church's Lambda Calculus, since the number of functional applications occurring in formula $P$ can be reduced to 1 by writing $(\lambda s.P).s$. It is understood from [10] that Dijkstra used the term "structure" to refer to the function $\lambda s.P$. If $P$ only depends on $s$, then the structure $\lambda s.P$ is an anonymous function without free variables.

Curry used the term "combinator" to refer to an anonymous function with no free variables [9]. However, no symbols other than $\lambda$ and bound variables were allowed in Curry's anonymous functions. Note that the structure $\lambda s.(\neg X.s \wedge Y.s) \vee Z.s$ uses the symbols $\neg$, $\wedge$ and $\vee$, which Curry did not allow. Therefore, the difference between a structure and a combinator is that the former are built on the language of applied lambda calculus (with variables and other symbols), while the latter are built on pure lambda calculus (with variables only).

Structures are very important for us. For example, the previous problem can be overcome using structures. The replacement of $x + 0$ by $0 + x$ in the formula $\varphi$ given by $(\forall x| : x + 0 = x)$, can be represent by defining $E$ as $(\forall x| : z.x = x)$ where "." is the function application. Then the formula $\varphi$ is rewritten as $E[z := \lambda x.x + 0]$, and the formula with the desired replacement is $E[z := \lambda x.0 + x]$. This can be verified as follows:

$$E[z := \lambda x.x + 0] \overset{\substack{\text{no free variables}\\ \text{x is not renamed}}}{\equiv} (\forall x| : (\lambda x.x + 0).x = 0) \overset{\text{functional application}}{\equiv} (\forall x| : x + 0 = x)$$

and

$$E[z := \lambda x.0 + x] \overset{\substack{\text{no free variables}\\ \text{x is not renamed}}}{\equiv} (\forall x| : (\lambda x.0 + x).x = 0) \overset{\text{functional application}}{\equiv} (\forall x| : 0 + x = x).$$

Note that the above replacement is logically correct, since the structures $\lambda x.x + 0$ and $\lambda x.0 + x$ are equal (because $x + 0$ and $0 + x$ are equal also).

Gries and Scheneider credit the above technique to Rutger Dijkstra [11]. Note that this is simpler than what is explained in the replacement metatheorem. All it takes is an equality of structures and a suitable $E$ with $z$ of type function to perform replacements. To ensure this, a logic was developed and mechanized where all formulas are structure equalities. Despite this, all formulas look like they do in LADM thanks to a user-friendly printing algorithm that acts as syntactic sugar.

Theorem provers like Coq [12], Isabelle [13], and Lean [14] are based on formal systems distinct from Calculational Logic. While some include modules for Calculational Logic, they inherit the complexities of their underlying logic. Therefore, it was decided to implement CalcLogic from scratch, building upon previous ideas that are compatible with the essence of the formal system. The assistant does not use Curry-Howard isomorphism [15] or a type-dependent system [16], but rather a much simpler type system (proving that complicated type systems are not necessary for mechanizing mathematical theories).

Unlike techniques based on Curry-Howard isomorphism, the logic in CalcLogic is neither constructivist nor intuitionist. Therefore, the reasoner legitimately processes proofs using the method of contradiction without needing to simulate it with metatheoretical techniques, as occurs in Coq and LEAN.

There are no other proof assistants, exclusively designed to mechanize Calculational Logic, that can load theories beyond those of LADM. CalcLogic has been able to load theories such as first-order Peano Arithmetic and Zermelo-Fraenkel Set Theory. The project aimed to make the assistant as universal as possible, so that proofs could be done in the widest possible range of mathematical theories. Currently, the assistant uses first-order quantification; however, its new functional foundations allow the construction of

schematic statements involving objects of arbitrary order (as long as they are not quantified). Thus, the web application is an assistant for first-order theories that allows writing axiom schemas with parameters of any order. Much of Mathematical Analysis and Arithmetic can be written within this framework.

In previous versions of CalcLogic, a complete proof could be done using only the mouse, so there was no need to learn a programming or markup language to use the application. A simple interface was found to maintain this in the new version of CalcLogic presented in this work. Everything can be done by means of the mouse even in Predicate Logic, despite the fact that this is typically a much more complicated theory than those CalcLogic could support (equational theories and Propositional Logic).

A predicate is a propositional function that receives a term and returns a proposition. The interface successfully conveys this functional structure of predicates. Also an intuitive way was found to manipulate the variables on which a predicate depends, its formula, and to distinguish between proposition, term, and predicate.

All operators, axioms and theorems from chapters 3, 4, 8 and 9 of LADM could be represented in CalcLogic and are printed on the screen in the same format as the book. Especially the generic quantifier, which is an operator that can be converted to $\forall$, $\exists$, $\Sigma$, $\prod$, etc. depending on the operator it receives. A simple and intuitive way was found for the user to pass an operator to the generic quantifier and use the quantification.

CalcLogic's calculational logic is even simpler than that of LADM, since it does not require the use of conditions written in metalanguage like $\neg occur('x',' P')$ (chapter 8 and 9 of [2]). That condition is not necessary, because the application can rename variables automatically.

A complete content of the Symbolic Logic course can be assessed with CalcLogic. The tool includes weakening and strengthening, which, unlike the previous versions, now also work within the quantifiers using the parity metatheorem. It also includes proofs by cases, by contradiction, by contrapositive, and by parts.

Two experiments were conducted, and the data suggest that students who had already know Calculational Logic, intuitively adapted to the interface. Students who were not performing well, showed an improvement in written exams after practicing with the application. This suggest that the interface is easy to use and the application helps students gain competencies.

## 3 Related Work

Calculational Logic was first described in the book [1] and later in [2] which popularized it. The book [2] is used in undergraduate computer logic courses at many universities. Previous references established the system's foundation but left some theoretical aspects open to a rigorous analysis. Subsequent foundational work took differing approaches, which led to misunderstandings that were ultimately clarified in the work [17].

There is no single standard set of inference rules for the system. For instance, while [18] proposes a system with four rules (Transitivity, Substitution, Leibniz, and Equanimity), the works [19, 20, 21, 22] include several types of Leibniz rules. Then the different variants of Leibniz's rule have been a topic of discussion [11], and the version used by [23] being slightly different from others. Moreover, in an attempt to add the Everywhere Operator [10, 24, 25, 26] the conclusion was that the Instantiation Rule should be removed. Following this same line, in the previous articles on CalcLogic [5, 6], the instantiation occurred at a meta-level. In this article, instantiation is an inference rule due to the new proof theory proposed.

The study of proof assistants is a part of Automated Theorem Proving, which itself is a subfield of a larger area called Automated Reasoning. The field of Automated Reasoning focuses on using a computer to perform reasoning based on the inference rules and syntax of a formal logic system.

The Automated Reasoning area is divided into the categories shown in Fig 1. Almost the entire core of CalcLogic has been reimplemented for this new version. This was due to the new theory of Calculational Logic, on which the new version is based. This theory explains that Calculational Logic can be encoded within the equational fragment of Lambda Calculus. Therefore, CalcLogic fits into the Equality Reasoning category; it is precisely an equational reasoner for anonymous functions (or structures). We know of no other assistant of this style for Calculational Logic.

Famous interactive proof assistants like Isabelle/HOL [13] (which uses its interactive window, PG Tips [28]) and Coq [12] all share a common characteristic: they are not based on Calculational Logic. The application presented in this work is also an interactive proof assistant, which is noteworthy for its unique foundation in this logic. The development of this new foundation is the additional contribution that distinguishes this extended version from the original work [7].

There are two proof checkers for Calculational Logic; CalcCheck stand alone [29], and CalcCheck web [30]. The first two applications are fundamentally the same, with the key difference being that one is an

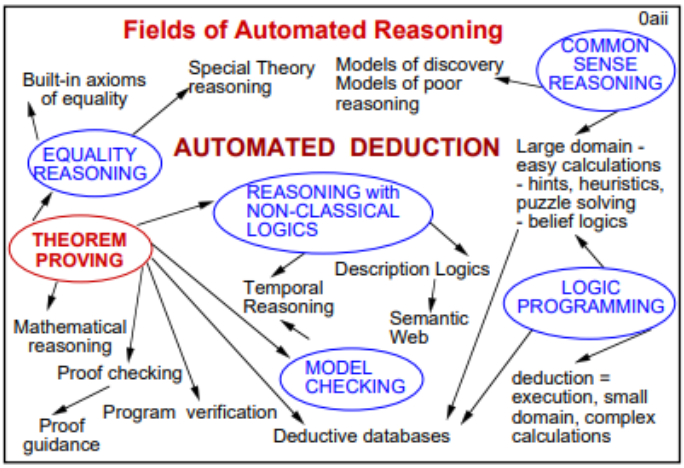



Figure 1: Sub areas of Automated Reasoning [27]

installable program and the other runs in a browser. However, a major distinction between them and the application in this work is their interface: the former requires proofs to be written in LaTeX for processing, whereas a CalcLogic user does not need to know any particular language, as all interactions are done with a mouse.

A proposed application in [31] aims to use the ideal features of Calculational Logic for automatic proof verification. This tool is designed to verify handwritten calculation proofs from a photograph. The authors explain how the theoretical properties of the logic are well-suited for carrying out this process.

The first educational use of CalcLogic occurred during the COVID-19 pandemic. At that time, it was used for Discrete Mathematics assessments as a way to either reduce or make student cheating evident. The results of this experiment are published in [4]. Then, to determine whether CalcLogic improves the competencies an experiment was conducted in [5], but no definitive conclusion was reached. Finally, the proof theory used for the previous version of CalcLogic and some algorithms are found in [6].

Another effort to use proof assistants in education, detailed in [32], involved the Coq assistant. In that study, students were required to learn a formal language to work with Coq, and then the coded proofs had to be translated into the language of mathematics books. The CalcLogic interface, in contrast, allows students to verify proofs directly using the same language in which math books write their formulas.

## 4 Paper Structure

The remainder of the work consists of two sections. The first of these (Section 5), develops a new proof theory that favors the implementation of the assistant and is closer to Dijkstra's functional ideas. In contrast, Section 6 shows the results of an educational experiment to validate the CalcLogic interface and answer research questions about learning Symbolic Logic. The Section 5 is much larger than the Section 6, however at the beginning of the former a summary of the structure of all its parts is made.

## 5 Proof theory for Calculational Logic of Dijkstra-Scholten

This section develops the language, inference rules, metatheorems, and proof methods of a new proof theory for the foundation of Calculational Logic. The new proof theory presented below uses the functional aspects expounded by Dijkstra, since every formula is encoded as an equality of structures.

The validity of the equivalence $(\forall x| : x + 0 = x) \equiv (\forall x| : 0 + x = x)$ is justified as the result of replacing $x + 0$ by $0 + x$ within $\forall$. But as explained earlier, to perform the replacement without modifying variables, the above formula should be understood as $(\forall x| : (\lambda x.x + 0).x = x) \equiv (\forall x| : (\lambda x.0 + x).x = x)$ where the replacement occurs between the functions $\lambda x.x + 0$ and $\lambda x.0 + x$. However, the language of LADM does not support anonymous functions, therefore the formula $(\forall x| : x + 0 = x) \equiv (\forall x| : 0 + x = x)$ hides the information regarding the functional application of structures (something that interested Dijkstra). Since we want to use the same language of LADM but taking into account the structures involved, it will be understood that $(\forall x| : x + 0 = x) \equiv (\forall x| : 0 + x = x)$ is the equivalence class of all the formulas that can be obtained from it, writing the involved structures in $\beta$ equivalent ways. For example, $(\forall x| : (\lambda x.x + 0).x = x) \equiv (\forall x| : (\lambda x.0 + x).x = x)$ would be a representative of the class since $(\lambda x.x + 0).x$ and $(\lambda x.0 + x).x$ are $\beta$ equivalent to $x + 0$ and $0 + x$ respectively.

Therefore, the language of Calculational Logic in this new proof theory is defined in two parts. The first is defined based on Applied Lambda Calculus (Subsection 5.1), and the second is defined as equivalence classes of first-level formulas (Subsections 5.1.2 and 5.1.3). The inference rules will also be defined in two parts: rules for the first language (Subsections 5.2), and then rules for the equivalence classes of formulas (Subsections 5.3). Following this, The one-step inference the user can perform from the interface (Subsection 5.4), the automatic renaming of variables (Subsection 5.5), the metatheorems and proof methods (Subsection 5.6 to 5.10) will be defined for the language based on equivalence classes.

### 5.1 Language

The language that CalcLogic can process is the language of Applied Lambda Calculus (Chapter 6 of [33]) with a simple-type system. The definition is as follows:

The set of variables consists of ASCII characters representing English letters (both uppercase and lowercase), i.e., the characters between "A-Z" and "a-z". These variables can denote a function.

The set of constants are symbols that denote specific objects (functions or not). This means that a symbol that denotes a specific operator is a constant in this language. For example $=_t$, $\neq_t$, $\equiv$, $\not\equiv$, $\Rightarrow$, $\Leftarrow$, $\wedge$, $\vee$, $\neg$ and $\Psi$ are constants (constant operators). Additionally, 0, 1, $\emptyset$, `tr`, $true$ and $false$ are constants.

The applicative language is being interpreted in the standard way, that is, $f\ x$ denotes the function application of $x$ on the function $f$. The function application is left-associative, that is, $f\ x\ y$ means $(f\ x)\ y$.

CalcLogic has a simple type system with two basic types, $p$ for propositions and $t$ for terms. Types can be composed with the function type constructor $\rightarrow$ to construct functional types.

The following constants in CalcLogic has the following types:

- $true$, `tr` and $false$ have type $p$
- $\emptyset$, 0 and 1 have type $t$
- $\neg$ has type $p \rightarrow p$
- $\equiv$, $\not\equiv$, $\Rightarrow$, $\Leftarrow$, $\wedge$, $\vee$ all have type $p \rightarrow p \rightarrow p$
- $+$ and $*$ have type $t \rightarrow t \rightarrow t$
- $=_t$ and $\neq_t$ have type $t \rightarrow t \rightarrow p$
- $\Psi$ has type $(x \rightarrow x \rightarrow x) \rightarrow (t \rightarrow p) \rightarrow (t \rightarrow x) \rightarrow x$, where $x$ is a type variable.

Some of these constants may be new to the reader. The symbol `tr` of type $p$ is synonymous of $true$, but is never displayed in the interface. It is used internally to store propositions without equivalences as equalities of structures, as we will explain in section 5.1.2. Similarly, since we established that $=$ denotes equalities of anonymous functions (in particular structures), the symbol $=_t$ was introduced to denote the equality of terms (of type $t$). However, the subscript $t$ is omitted when printed to the user. The symbol $\Psi$ will be explained in the next section.

A well-formed phrase (WFP) in the language can be constructed with the following rules:

- A variable is a WFP.
- A constant of type $p$ or $t$ is a WFP.
- If $p_1$ and $p_2$ are WFPs of type $r \rightarrow s$ and $r$, respectively, then $p_1\ p_2$ is a WFP of type $s$.
- If $p_1$ is a WFP of type $s$ and $x$ is a variable of type $r$, then $\lambda x.p_1$ is a WFP of type $r \rightarrow s$.

**Note.** *The usual rule for determining the type of an abstraction is "$x : r, \Gamma \vdash p_1 : s$ then $\Gamma \vdash \lambda x.p_1 : r \rightarrow s$", which is context-sensitive. In CalcLogic, an explicit context is not used to determine the type of $x$; instead, it is deduced from the formula where it occurs. As will be seen later, $\lambda x.p_1$ only occurs as an argument of an operator that bounds variables (such as $\forall$ or $\exists$) or occurs in the form $\lambda y, .., x, .., z.p_1 = \lambda y, .., x, .., z.p_2$ where $x$ occurs in $p_1$ or $p_2$. The algorithm for deducing the type of $x$ in each case will be published in a future paper. CalcLogic throws an error when attempting to use the phrase $\lambda x.p_1$, where the type of $x$ cannot be deduced. However, we have not found any formulas in LADM theories where this occurs (nor in the other theories we have experimented with, such as ZF set theory or First Order Arithmetic).*

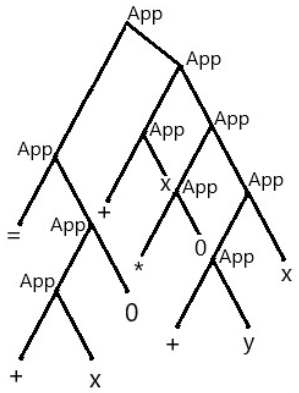



Figure 2: Binary tree representing the term $= \ (+\ x\ 0)\ (+\ x\ (*\ 0\ (+\ y\ x)))$.

For example, if $x$, $y$ are of type $t$, then $(=_t \ x\ y)$ is of type $p$ (proposition), whereas $(\lambda x.\lambda y.\ =_t \ x\ y)$ is of type $t \to t \to p$, that is, a propositional function of two variables. The predicates in CalcLogic are propositional functions (from terms to propositions), inspired by Church [8].

Although this is an applicative language, to improve readability and align with conventional mathematical practices, we will present phrases using the infix notation if it is convenient. For example, instead of writing $\wedge\ Q\ z, \Rightarrow \ Q\ P, \wedge\ (\Rightarrow \ Q\ P)\ z, =_t \ x\ y$ and $\in \ \mathbb{N}\ (+\ x\ 2)$ we can write $z \wedge Q$, $P \Rightarrow Q$, $z \wedge (P \Rightarrow Q)$, $y =_t x$ and $2 + x \in \mathbb{N}$, respectively. In addition, CalcLogic uses a printing algorithm, published in Section 6.1 of [6], that converts the applied lambda-calculus notation into infix notation so that the formulas look like those in LADM.

The function application denoted as $fx$ is represented in the CalcLogic backend as a binary tree. This tree consists of a parent node, labeled with the App operator, and two children labeled $f$ and $x$. For example, the phrase $= \ (+\ x\ 0)\ (+\ x\ (*\ 0\ (+\ y\ x)))$ is represented as the tree in Figure 2.

The notation above is the one traditionally used in Lambda Calculus. However, in Calculational Logic, the function application of the argument $x$ to the function $f$ is denoted by $f.x$ according to [1, 2]. This latter notation is what is printed to the user in CalcLogic and is the one that will be used in this work with formulas in the language of Calculational Logic.

### *5.1.1 The Generic Quantifier*

For Church and Curry [8, 34], the quantifiers $\forall$ and $\exists$ were restricted to a predicate that determines the range of quantification. The same idea was adopted by Dijkstra and Scholten in their book [1], and then by Gries and Scheneider [2], where quantifiers are expressed in the form $(\forall x|R : P)$ and $(\exists x|R : P)$. $R$ is the range in which the quantification is performed and $P$ is the quantified predicate. For Church [8] (and also for Frege [35]) a predicate such that $R$ and $P$ were propositional functions, i.e functions of type $t \to p$. For example, $x =_t x$ is a proposition, while $\lambda x.x =_t x$ is a predicate. In the first, $x$ is fixed but unknown ($x$ is a parameter), so the truth value is fixed but unknown. In the second, the truth value is not fixed and depends on the value of $x$ in the function. Quantifiers can only be applied to predicates and not to propositions. In this way, the formula $(\forall x|R : P)$ was written by Church [8] as $\forall\ R\ P$ where $\forall : (t \to p) \to (t \to p) \to p$ and $R, P : t \to p$, or $\forall\ (\lambda x^t.R)\ (\lambda x^t.P)$ if $R$ and $P$ are propositions (of type $p$).

In Chapter 8, LADM presents general quantification rules for any associative and commutative binary operator $\star$, with an identity element (forming a commutative monoid). The quantifier for such an operator is denoted $(\star x|R : P)$.

CalcLogic implements this with a special quantification operator $\Psi$ of type $(x \to x \to x) \to (t \to p) \to (t \to x) \to x$, which operates on a binary operator of a commutative monoid (e.g., $\vee$, $\wedge$), a range predicate, and a quantified expression. $\Psi$ is used to make a quantification. For example, the application $\Psi \vee$ corresponds to $\exists$, the application $\Psi \wedge$ corresponds to $\forall$, the formula $\Psi \ \wedge \ (\lambda x.R)\ (\lambda x.P)$ corresponds to $(\forall x|R : P)$ and $\Psi \ + \ (\lambda x.R)\ (\lambda x.P)$ corresponds to $(+x|R : P)$ (or in sum notation $\sum_R P$). Additionally, for readability, when $R$ is *true*, then CalcLogic prints $(\star x| : P)$ for any $\star$ operator.

### *5.1.2 Well Formed Formula (WFF)*

As already mentioned, all formulas in CalcLogic are equalities of structures. The LADM formulas of the form $P =_t Q$ and $P \equiv Q$ are encoded as $\lambda \overline{x}.P = \lambda \overline{x}.Q$, where $\overline{x}$ is the list of all free variables that occur in

$P$ and $Q$, and $=$ is the equality of functions. For example, the formula $true \equiv (q \equiv q)$ of LADM is encoded as $\lambda q.true = \lambda q.q \equiv q$.

**Definition.** *The formula $\lambda\overline{x}.P = \lambda\overline{x}.Q$ is a structure equality (SE), where $P$ and $Q$ are WFP (well-formed phrases), $\lambda\overline{x}.P$ has the same type of $\lambda\overline{x}.Q$ and there are no occurrences of free variables in $\lambda\overline{x}.P$ and $\lambda\overline{x}.Q$. In an SE the order of the variables in the list $\overline{x}$ is determined by their order of occurrence when searching for them using DFS in the term $P = Q$. This forces the order in the list $\overline{x}$ to not depend on the variable names.*

For example, in the case where $P$ and $Q$ are the terms $+\ x\ (*\ 0\ (+\ y\ x))$ and $+\ x\ 0$ respectively, the equality $P = Q$ is actually represented in application notation as $=\ (+\ x\ 0)\ (+\ x\ (*\ 0\ (+\ y\ x)))$, using the tree data structure in the Figure 2. When performing a DFS search in that tree, $x$ is found first and then $y$, therefore the related structure equality is $\lambda x, y.+\ x\ (*\ 0\ (+\ y\ x)) = \lambda x, y.+\ x\ 0$. The order in which the variables are abstracted is important, since $\lambda x, y.+\ x\ (*\ 0\ (+\ y\ x))$ is not the same function as $\lambda y, x.+\ x\ (*\ 0\ (+\ y\ x))$, for this reason a unique way of ordering the list of abstractions $\lambda$ in an SE was established. The list is ordered in the order in which DFS finds the variables in the term.

If a formula is not an equality, it can also be coded as an SE. For example, $p \vee \neg p$ can be encoded as $\lambda p.p \vee \neg p = \lambda p.\mathtt{tr}$, where `tr` represents "true". In general, if $P$ is a LADM formula that is not of the form $e_1 \equiv e_2$ or $e_1 =_t e_2$, it can be encoded as $\lambda\overline{x}.P = \lambda\overline{x}.\mathtt{tr}$.

**Definition.** *A Well Formed Formula (WFF) in Calculational Logic is an equivalence class of SE, where all of them are $\beta$ equivalent to each other. The CalcLogic backend does not handle the whole class, but one representative of the class. The representative is the $\beta$ normal form of any element of the class. This class is denoted as $[P = Q]$ when its representative is of the form $\lambda\overline{x}.P = \lambda\overline{x}.Q$.*

The usual notation for writing a WFF in LADM is the one obtained by taking a representative of the class, and then applying the printing algorithm of [6] (section 6.1). This algorithm is only applicable when $\lambda\overline{x}.P = \lambda\overline{x}.Q$ is in $\beta$ normal form. Therefore, representatives of the classes $[P = Q]$ can always be printed for the user like LADM with this algorithm. That is why it was chosen to work with these representatives. For example the representative of the class $[(\forall x|\ :\ (\lambda x.x + 0).x\ =\ x)\ =\ (\forall x|\ :\ (\lambda x.0 + x).x\ =\ x)]$ is $(\forall x| : x+0 = x) = (\forall x| : 0+x = x)$ and the algorithm of [6] prints this as $(\forall x| : x+0 = x) \equiv (\forall x| : 0+x = x)$ (which is the format of LADM).

On the other hand, the LADM's infix notation hides functional aspects of the formula. For example, the formula $2+x \in \mathbb{N}$ in infix notation may hide the applicative structure with which the evaluation of the terms is obtained. For example, it can be $\in\ \mathbb{N}\ (+\ x\ 2)$ or $\in\ \mathbb{N}\ ((\lambda y.+\ x\ y)\ 2)$ or $\in\ \mathbb{N}\ ((\lambda y, x.+\ x\ y)\ 2\ x)$. The idea is that the formula $2 + x \in \mathbb{N}$, which is displayed to the user, represents an entire class of all possible applicative ways to write this formula, that is, the equivalence class $[2 + x \in \mathbb{N} = tr]$. For Dijkstra and Scholten, all these variations of the applicative structures were important in their proofs (even if they were not explicit); in fact, they named them "structures" [10].

The function equality $=$ cannot be nested, for example $[(P = Q) = R]$ is not a WFF. This decision was made to prevent the user from explicitly using higher-order functions without any limitations. The type of functions visible to the user must be specified in the configuration panel. Controlling function orders prevents the existence of logical inconsistencies.

To simplify the notation, a WFF $[P = \mathtt{tr}]$ can be abbreviated as $[P]$. For ease of reading, in this work $P$, $e_1$ and $e_2$ can be written in infix notation inside the brackets "[.]". Recall that in CalcLogic's backend they are written in applicative notation. For example, it is permissible to write $[a + b =_t b + a]$ instead of $[(=_t\ \ (+\ a\ b)\ (+\ b\ a)) = \mathtt{tr}]$ (in the formula, the type of the equality $=_t$ is limited to terms of type $t$. To include the equality of functions, the administrator must add an equality symbol with the specific function type to compare).

CalcLogic displays a WFF of the form $[P = \mathtt{tr}]$ as $P$, while a WFF of the form $[e_1 = e_2]$ is displayed as $e_1 \equiv e_2$ if $e_1$, $e_2$ are of type $p$, or $e_1 = e_2$ if $e_1$, $e_2$ are of type $t$.

It is also possible to understand the notation [.] as an operator. The following will explain

**Definition.** *The "everywhere operator" is defined as the operator that takes a term from the applied lambda calculus $P$ and returns the WFF $[P]$. That is, if $P$ is of the form $t_1 = t_2$, then the everywhere operator returns $[t_1 = t_2]$, and otherwise it returns $[P = \mathtt{tr}]$.*

*The name "everywhere" comes from the fact that since $[t_1 = t_2]$ means a term $\beta$-equivalent to $\lambda\overline{x}.t_1 = \lambda\overline{x}.t_2$, this is equivalent to the equality $t_1 = t_2$ being true for all values of its variables $\overline{x}$ (everywhere). Also if $t_2$ is `tr`, then $[t_1]$ is equivalent to $t_1$ being true for all values of its variables (everywhere). The name "everywhere operator" and the notation [.] come from [1], although Dijkstra and Scholten defined the same thing without using anonymous functions or equivalence classes.*

### 5.1.3 *Axiom Schemes*

Although the proposed quantification is of first-order (only variables of type $t$ can be quantified), the language allows objects of any order, as long as they are not quantified. For example, by writing $\neg(f(\lambda x.\neg x))$, it follows that $f$ is a variable of type $(p \to p) \to p$, that is, a higher-order function. The quantifiers $\forall$ and $\exists$ cannot bound a $f$ of this type.

However, in the metatheory it is true that:

$$\lambda\overline{x}.t_1 = \lambda\overline{x}.t_2 \text{ iff } (t_1 = t_2 \text{ for all } \overline{x})$$

where $\overline{x}$ can be of any type. In this way, it can be assumed that all free variables in $t_1 = t_2$ are universally quantified in a formula like $[t_1 = t_2]$ regardless of their type.

For example, the separation axiom scheme of ZF set theory

$$(\exists A| : (\forall x| : x \in A \equiv x \in U \wedge P.x.U))$$

can be written in applicative notation as:

$$[\Psi \ \vee \ (\lambda A.true) \ (\lambda A.\Psi \ \wedge \ (\lambda x.true) \ (\lambda x. \equiv \ (\wedge \ (P \ x \ U) \ (\in \ U \ x)) \ (\in \ A \ x)))]$$

The variable $P$, which is a predicate that depends on $x$ and $U$, would be universally quantified in the metatheory although it is not of base type $t$.

This observation leads to the following conclusion regarding the proposed proof theory. Although CalcLogic can only work with first-order theories, axiom schemes can be loaded using objects of any order, provided they are not quantized.

## 5.2 Inference Rules of the CalcLogic Assistant

The inference rules used in CalcLogic differ a bit from those presented in LADM due to our new implementation using anonymous functions and structures. In the following, we will mention the rules that the assistant uses, which are some from Lambda Calculus's equational fragment. The symbol $\Gamma$ denotes the set of all possible premises used to make an inference, i.e. the axioms of the theory and the theorems already proven. $\Gamma \vdash t_1 = t_2$ means "equality $t_1 = t_2$ can be derived from premises $\Gamma$".

### 5.2.1 *Symmetry*

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1} \text{ S}$$

The right label S identifies the applied rule.

### 5.2.2 *Projection of Premise*

If $t$ is an equality in $\Gamma$ labeled with the identifier id:

$$\frac{}{\Gamma \vdash t} \text{ P id}$$

### 5.2.3 *Transitivity*

$$\frac{\Gamma \vdash t_1 = t_2 \qquad \Gamma \vdash t_3 = t_4}{\Gamma \vdash t_1 = t_4} \text{ T, when } t_2 =_\beta t_3$$

where $=_\beta$ is the $\beta$ equivalence of Lambda Calculus $\Lambda$.

### 5.2.4 *Application*

If $(\lambda z.E)t_1$ and $(\lambda z.E)t_2$ are well-typed and the type is the same:

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda z.E)t_1 = (\lambda z.E)t_2} \text{ A}$$

Note that the reflexivity statement $E = E$ is obtained using this A rule with $z \notin E$.

#### 5.2.5 Leibniz's Rule

If $E$ is of type $t \to p$ and $\overline{x}$ is the free variables list that occurs in $t_1$ and $t_2$:

$$\frac{\Gamma \vdash \lambda E.\lambda \overline{x}.E\ t_1 = \lambda E.\lambda \overline{x}.E\ t_2}{\Gamma \vdash \lambda \overline{x}.t_1 = \lambda \overline{x}.t_2}\ \mathrm{L}$$

The underlying concept of Leibniz's Rule is that if everything that can be said about $t_1$ can equally be said about $t_2$, then $t_1$ and $t_2$ are equal. The symbol $E$ is a variable, it can be any predicate, therefore anything can be said about $t_1$ and $t_2$ in the formula. This Leibniz's rule is not the same as LADM. What in LADM is called Leibniz's Rule, in this work is preferred to be called Replacement Rule (see next section).

These rules are some of the equational fragment of Lambda Calculus, but we only need these for the assistant.

These rules are combined to form derivation trees (or inference trees). For each sequence of inferences made by the user, a derivation tree is constructed in the application's backend. An incorrect inference by the user would result in a tree that does not comply with the aforementioned inference rules. In that case, an exception is thrown and propagated to the user interface, resulting in the message "Invalid inference".

### 5.3 Inference Rules of Calculational Logic

This section proves that inference rules on formulas of the form $[P = Q]$ can be derived from the rules of the previous section. These new rules correspond to inference rules of the Calculational Logic according to LADM and [25]. In this sense, this section provides a theoretical overview of Calculational Logic and its connection to Lambda Calculus.

In our Calculational Logic, all statements take the form $[P = Q]$, and inference rules operate by transforming statements of this form into others of the same form. However, the above rules do not operate on equivalence classes; therefore, the rules in this section are derivable in the following sense. Every rule $R$ of Calculational Logic can be written as a derivation tree scheme $T$ using the previous rules. The tree scheme is such that if we take representatives from the premises of $R$ and place them in the premises of $T$, then root of $T$ belongs to the class of the root of $R$.

This property was used to implement the next inference rules as the tree schemas mentioned. For each inference in the tree, CalcLogic computes the $\beta$-normal form of the root to represent a WFF at each step.

The Calculational Logic rules and their derivations are the following:

#### 5.3.1 Projection of Premise

This rule is the same as in the previous section but applied to a WFF. To write the proofs in strings and save them in a database, the derivation trees are written in applicative notation using codes for each rule. The code of the projection rule is $\mathrm{P}^{id}$.

#### 5.3.2 Symmetry

$$\frac{\Gamma \vdash [P = Q]}{\Gamma \vdash [Q = P]}\ \mathrm{S}$$

This is the same symmetry rule as that in the previous section, but applied to a WFF. If $\nabla$ encodes the derivation of $\Gamma \vdash [P = Q]$, then this rule is encoded as $(\mathrm{S}\ \nabla)$.

#### 5.3.3 Transitivity

$$\frac{\Gamma \vdash [P = Q] \qquad \Gamma \vdash [Q = R]}{\Gamma \vdash [P = R]}\ \mathrm{T}$$

Notice that there are hidden lambda abstractions $\lambda\overline{x}$ in statements $[P = Q]$ and $[Q = R]$, where $\overline{x}$ denotes the list of free variables, like $\overline{x} = x_1, x_2, \ldots, x_n$. However, that list could be different for each statement, even for the $Q$'s on each side. For example, $\neg false = true$ is representative of $[\neg false = true]$, while $\lambda q.true = \lambda q.q \equiv q$ is representative of $[true = q \equiv q]$. Therefore, variable lists in $\lambda$-abstractions are not the same for the two $true$ expressions involved in $[\neg false = true]$ and $[true = q \equiv q]$.

If $\lambda\overline{w}.P = \lambda\overline{w}.R$ is the representative of $[P = R]$, then this rule can be derived from the inference rules of the previous section in the following way:

$$\dfrac{\dfrac{\Gamma \vdash \lambda\overline{x}.P = \lambda\overline{x}.Q}{\Gamma \vdash (\lambda z.\lambda\overline{w}.z\ \overline{x})(\lambda\overline{x}.P) = (\lambda z.\lambda\overline{w}.z\ \overline{x})(\lambda\overline{x}.Q)}\,\mathrm{A} \quad \dfrac{\Gamma \vdash \lambda\overline{y}.Q = \lambda\overline{y}.R}{\Gamma \vdash (\lambda z.\lambda\overline{w}.z\ \overline{y})(\lambda\overline{y}.Q) = (\lambda z.\lambda\overline{w}.z\ \overline{y})(\lambda\overline{y}.R)}\,\mathrm{A}}{\Gamma \vdash (\lambda z.\lambda\overline{w}.z\ \overline{x})(\lambda\overline{x}.P) = (\lambda z.\lambda\overline{w}.z\ \overline{y})(\lambda\overline{y}.R)}\,\mathrm{T}$$

Since $(\lambda z.\lambda\overline{w}.z\ \overline{x})(\lambda\overline{x}.Q)$ and $(\lambda z.\lambda\overline{w}.z\ \overline{y})(\lambda\overline{y}.Q)$ are $\beta$-equivalent to $\lambda\overline{w}.Q$, we have reached the root of the previous tree whose $\beta$-normal form is $\lambda\overline{w}.P = \lambda\overline{w}.R$, who is the representative of $[P = R]$. Note that $z\ \overline{x}$ denotes $z\ x_1\ x_2 \ldots x_n$.

If $\nabla_1$ and $\nabla_2$ encode the derivations of $\Gamma \vdash [P = Q]$ and $\Gamma \vdash [Q = R]$, respectively, then this rule is encoded as $(\nabla_1\ \nabla_2)$.

### *5.3.4 Equanimity*

$$\dfrac{\Gamma \vdash [P = Q] \qquad \Gamma \vdash [P]}{\Gamma \vdash [Q]}\,\mathrm{E}$$

It is actually derived in this way:

$$\dfrac{\dfrac{\Gamma \vdash [P = Q]}{\Gamma \vdash [Q = P]}\,\mathrm{S} \qquad \Gamma \vdash [P = \mathtt{tr}]}{\Gamma \vdash [Q = \mathtt{tr}]}\,\mathrm{T}$$

If $\nabla_1$ and $\nabla_2$ encode the derivations of $\Gamma \vdash [P = Q]$ and $\Gamma \vdash [P]$, respectively, then this rule is encoded as $(\nabla_1\ \nabla_2)$.

### *5.3.5 Replacement Rule*

$$\dfrac{\Gamma \vdash [P = Q]}{\Gamma \vdash [E^z_P = E^z_Q]}\,\mathrm{R}$$

The formula scheme $E^z$ is a formula $E$ in which some of its sub-formulas may have been denoted by placeholder $z$. The notation $E^z_P$ denotes formula scheme $E^z$ with $z$ replaced by $P$ (without renaming variables).

To infer $[E^z_P = E^z_Q]$ from $[P = Q]$, whose representative is $(\lambda\overline{x}.P) = (\lambda\overline{x}.Q)$, where $\overline{x}$ and $\overline{y}$ are the list of free variables that occur in $P = Q$ and $E^z_P = E^z_Q$, respectively, then we can use the Application rule with $\lambda w.\lambda\overline{y}.E^z_{w\ \overline{x}}$ in the following way:

$$\dfrac{\Gamma \vdash \lambda\overline{x}.P = \lambda\overline{x}Q}{\Gamma \vdash (\lambda w.\lambda\overline{y}E^z_{w\ \overline{x}})\ (\lambda\overline{x}.P) = (\lambda w.\lambda\overline{y}E^z_{w\ \overline{x}})\ (\lambda\overline{x}.Q)}\,\mathrm{A}$$

The *beta*-reductions of the previous inference tree root can be reduced as

$$\begin{array}{ll}
 & (\lambda w.\lambda\overline{y}E^z_{w\ \overline{x}})\ (\lambda\overline{x}.P) = (\lambda w.\lambda\overline{y}E^z_{w\ \overline{x}})\ (\lambda\overline{x}.Q) \\
\iff & \\
 & (\lambda\overline{y}E^z_{w\ \overline{x}})[w := (\lambda\overline{x}.P)] = (\lambda\overline{y}E^z_{w\ \overline{x}})[w := (\lambda\overline{x}.Q)] \\
\iff & \quad \langle \lambda\overline{x}.P \text{ and } \lambda\overline{x}.Q \text{ do not have free variables,} \\
 & \qquad \text{the substitutions are made without renaming variables} \rangle \\
 & \lambda\overline{y}E^z_{(\lambda\overline{x}.P)\ \overline{x}} = \lambda\overline{y}E^z_{(\lambda\overline{x}.Q)\ \overline{x}} \\
\iff & \\
 & \lambda\overline{y}E^z_P = \lambda\overline{y}E^z_Q \\
\in & \\
 & [E^z_P = E^z_Q]
\end{array}$$

If $\nabla$ encodes the derivation of $\Gamma \vdash [P = Q]$, then this rule is encoded as $(\mathrm{R}^{\lambda z.E^z}\ \nabla)$.

The Replacement Rule is called Leibniz's Rule in LADM. The name Replacement Rule in this work is because the name Leibniz's Rule was already used for another rule.

As shown above, the use of SE in the derivation of this rule is quite elegant. Gries and Scheneider credit this technique to Rutger Dijkstra [11]. Our idea of formalizing Calculational Logic using only SE is inspired on this technique.

*5.3.6 Instantiation*

$$\frac{\Gamma \vdash [P = Q]}{\Gamma \vdash [P[\overline{x} := \overline{Exp}] = Q[\overline{x} := \overline{Exp}]]}\ \mathrm{I}$$

Since the representative of $[P = Q]$ had no free variables and there always exists a list $\overline{Exp'}$ such that $(P = Q)[\overline{x} := \overline{Exp}]$ is the same as $(P = Q)[\overline{x'} := \overline{Exp'}]$, with $\overline{x'}$ all the free variables of $P = Q$, then the above rule can be derived by taking a representative of $[P = Q]$ and using Application rule with $E$ as $\lambda \overline{y}.z\ Exp'_1 \cdots Exp'_n$ with $\overline{Exp'} = Exp'_1, \cdots, Exp'_n$ and $\overline{y}$ the list of free variables that occur in $P[\overline{x} := \overline{Exp}] = Q[\overline{x} := \overline{Exp}]$.

If $\nabla$ encodes the derivation of $\Gamma \vdash [P = Q]$, then this rule is encoded as $(\mathrm{I}^{[\overline{x}:=\overline{Exp}]}\ \nabla)$. (Note that $Q$ could be `tr`)

This rule did not exist in [6] because in that work the axioms were considered schemas. The schema instantiation occurred at the metalevel in [6]. According to the theory developed here, instantiation is a proper inference rule.

*5.3.7 Leibniz's Rule*

$$\frac{\Gamma \vdash [E.t_1 = E.t_2]}{\Gamma \vdash [t_1 = t_2]}\ \mathrm{L}$$

This rule is derived using the Leibniz's rule of section 5.2 with the canonic representative of $[E.t_1 = E.t_2]$ (the $\beta$ normal form of any element of the class) as premise.

*5.3.8 Modus Ponens*

$$\frac{\Gamma \vdash [P \Rightarrow Q] \qquad \Gamma \vdash [P]}{\Gamma \vdash [Q]}\ \mathrm{M}$$

It is derived from the axioms of $\Rightarrow$ and the above rules. However, it would be expensive to make the derivation each time it is needed, so CalcLogic implements it as an atomic rule. Anyway, the derivation of this rule is shown below by parts:

1. The following tree depends on premise $[P]$. It will be abbreviated as $\nabla([P])$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [true \wedge p = p]}\,\mathrm{P}}{\Gamma \vdash [true \wedge Q = Q]}\,\mathrm{I}}{\Gamma \vdash [Q = true \wedge Q]}\,\mathrm{S} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\nabla_1}{\Gamma \vdash [P]}}{\Gamma \vdash [P = true]}\,\mathrm{M1}(\nabla_1)}{\Gamma \vdash [true = P]}\,\mathrm{S}}{\Gamma \vdash [true \wedge Q = P \wedge Q]}\,\mathrm{R}}{\Gamma \vdash [Q = P \wedge Q]}\,\mathrm{T} \qquad \dfrac{\dfrac{}{\Gamma \vdash [p \wedge (p \Rightarrow q) = p \wedge q]}\,\mathrm{P}}{\Gamma \vdash [P \wedge (P \Rightarrow Q) = P \wedge Q]}\,\mathrm{I}}{\Gamma \vdash [P \wedge Q = P \wedge (P \Rightarrow Q)]}\,\mathrm{S}}{\Gamma \vdash [Q = P \wedge (P \Rightarrow Q)]}\,\mathrm{T}$$

The label $\mathrm{M1}(\nabla_1)$ says that the inference is made by Meta Theorem 1 of section 5.6.

2. The following tree depends on premise $[P]$. It will be abbreviated as $\nabla'([P])$:

$$\dfrac{\dfrac{\dfrac{\nabla([P])}{\Gamma \vdash [Q = true \wedge (P \Rightarrow Q)]}\,\mathrm{T} \qquad \dfrac{\dfrac{\dfrac{\nabla_1}{\Gamma \vdash [P]}}{\Gamma \vdash [P = true]}\,\mathrm{M1}(\nabla_1)}{\Gamma \vdash [P \wedge (P \Rightarrow Q) = true \wedge (P \Rightarrow Q)]}\,\mathrm{R}}{\Gamma \vdash [Q = P \Rightarrow Q]}\,\mathrm{T} \qquad \dfrac{\dfrac{}{\Gamma \vdash [true \wedge p = p]}\,\mathrm{P}}{\Gamma \vdash [true \wedge (P \Rightarrow Q) = (P \Rightarrow Q)]}\,\mathrm{I}}{\Gamma \vdash [P \Rightarrow Q = Q]}\,\mathrm{S}$$

3. The following tree, which depends on the premises $[P]$ and $[P \Rightarrow Q]$, is a derivation of the Modus Ponens inference rule:

$$\frac{\nabla'([P]) \qquad \dfrac{\nabla_2}{\Gamma \vdash [P \Rightarrow Q]}}{\Gamma \vdash [Q]}\ \mathrm{E}$$

The applicative tree notation is useful for encoding proofs in a notation that can be parsed. For example, if $[true \wedge p = p]$, $[p \wedge (p \Rightarrow q) = p \wedge q] \in \Gamma$ with IDs 1 and 2, respectively, then the entire derivation tree above can be encoded as:

$$\mathrm{S}(\mathrm{S}(\mathrm{I}^{[p:=Q]}\mathrm{P}^1)(\mathrm{R}^{\lambda z.\wedge Qz}(\mathrm{S}\ \mathrm{M}_1(\nabla_1)))(\mathrm{S}(\mathrm{I}^{[p,q:=P,Q]}\mathrm{P}^2))(\mathrm{R}^{\lambda z.\wedge(\Rightarrow QP)z}\mathrm{M}_1(\nabla_1))(\mathrm{I}^{[p:=\Rightarrow QP]}\mathrm{P}^1))\nabla_2$$

However, since Modus Pones was decided to be an atomic inference rule, the application of the rule is saved in the database as the simplified string $(\nabla_1\nabla_2)$. The symbols $\nabla_1$ and $\nabla_2$ are derivation trees with root $[P \Rightarrow Q]$ and $[P]$ respectively.

It is important to note that the beta reductions involved in the derivation of all inference rules are convenient in several aspects. For that reason, these rules are derived whenever they are used in CalcLogic (except modus ponens). For example, section 5.5 explains how these beta reductions can rename a quantifier's dummy variable when necessary.

To conclude this section, we define that $[P]$ is a theorem in the context of premises $\Gamma$ if and only if $\Gamma \vdash [P]$. That is, when there exists a derivation tree, using $\Gamma$, whose root is $[P]$.

### 5.4 One-step Inference from User's Point of View

From the user's point of view, the most basic inferences involve the combination of up to four rules of Calculational Logic: Projection, Instantiation, Replacement and Symmetry, in that order. Figure 3 presents the instantiation form of the CalcLogic interface, where the user can select a statement from $\Gamma$ (axioms or previously proven theorems), specify a substitution to instantiate the statement, and choose the $E^z$ term for the replacement rule. To align with LADM terminology, this last field is displayed to the user under the label "Leibniz" instead of "Replacement."

A button panel simplifies the input of mathematical expressions in the latter two fields of the form. Sections 4.2.1–4.2.4 of [4] and sections 5.3.1–5.3.3 of [6] describe various mechanisms that assist in filling these fields. Notably, if needed, the letter $z$ in $E^z$ can be modified, as the field remains editable within the form.

If the user selects the statement $[P = Q]$, which is labeled as $id$, the substitution $[\overline{x} := \overline{Exp}]$ and a formula $\varphi$ for $E^z$, the following tree is generated in the backend:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [P = Q]}\ \mathrm{P\ id}}{\Gamma \vdash [P[\overline{x} := \overline{Exp}] = Q[\overline{x} := \overline{Exp}]]}\ \mathrm{I}}{\Gamma \vdash [E^z_{P[\overline{x}:=\overline{Exp}]} = E^z_{Q[\overline{x}:=\overline{Exp}]}]}\ \mathrm{R}$$

(This tree is denoted as $\mathrm{R}^{\lambda z.E^z}(\mathrm{I}^{[\overline{x}:=\overline{Exp}]}\mathrm{P}^{id})$ in applicative notation).

The previous tree is displayed to the user in the following vertical way:

$$\begin{array}{ll} & A_1 \\ eq & \langle \text{st (id) with } \overline{x} := \overline{Exp} \text{ and } E^z : \varphi \rangle \\ & A_2 \end{array}$$

Where $A_1$ and $A_2$ are the formulas resulting from the calculation of $E^z_{P[\overline{x}:=\overline{Exp}]}$ and $E^z_{Q[\overline{x}:=\overline{Exp}]}$, respectively, $eq$ is the symbol $=$ if $A_1$ and $A_2$ are of type $t$, or the symbol $\equiv$ if they are of type $p$. The comment enclosed between $\langle\ \rangle$ is called hint, and the phrase $st$ is an abbreviation for "statement".

An example of how inference is made in CalcLogic is shown in Figure 4. This figure shows the result of instantiating the substitution and replacement forms according to Figure 3.

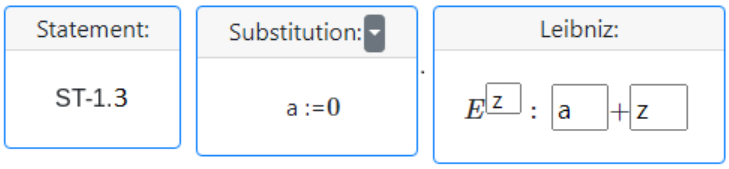



Figure 3: Forms for instantiating a statement and specifying a replacement.

It is also possible that the user leaves the substitution or replacement fields empty, in which case the inference tree would not have the correspondent rules. All the resulting tree combinations are:

Leaving both fields empty, the following tree is generated in the backend:

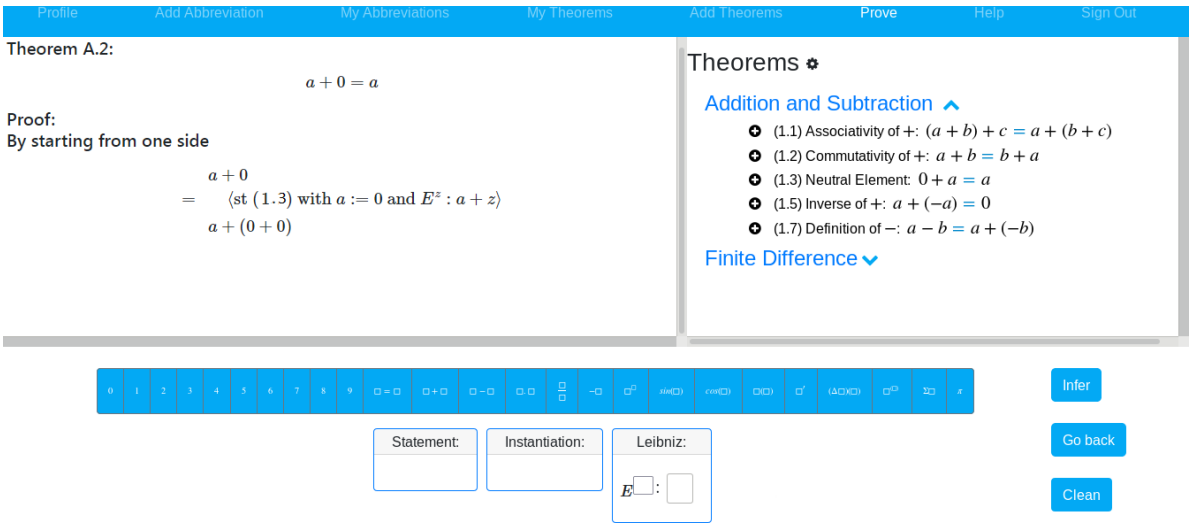



Figure 4: Result inference of filling the form of Fig. 3. The symbol *eq* is $=$ because $a+0$ and $a+(0+0)$ are of type $t$.

$$\frac{}{\Gamma \vdash [P = Q]}\ \mathrm{P\ id}$$

This tree is denoted as $\mathrm{P}^{id}$ in applicative notation. It is displayed to the user in the following vertical notation:

$P$
*eq* ⟨st (id)⟩
$Q$

If $Q$ is `tr` in the previous tree and $P$ is of the form $A_1$ *op* $A_2$, then the display changes in this way:

$A_1$
*op* ⟨st (id)⟩
$A_2$

(The most common use of this type of inference is when *op* is $\Rightarrow$ or $\Leftarrow$ to weaken or strengthen a formula, but this type of inference is blocked if the appropriate proof method is not selected. We will call inferences of this kind *op*-inference).

An example of how CalcLogic displays these two types of inferences can be found in Figures 5 and 6 respectively.

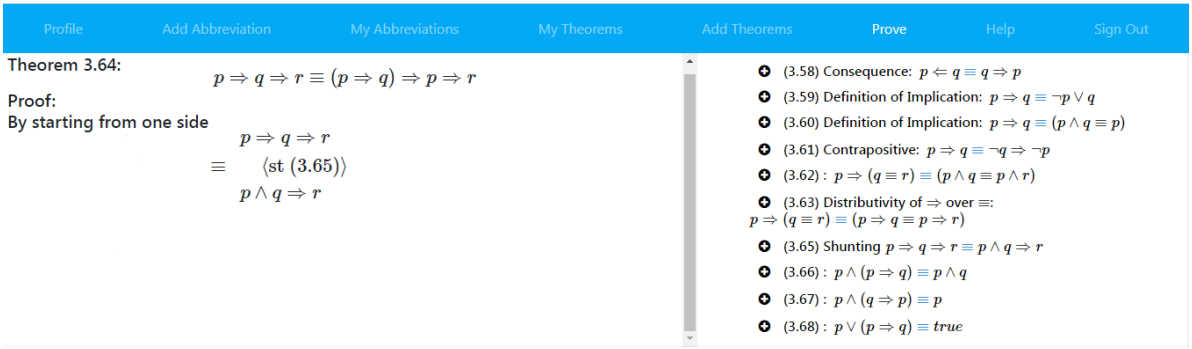



Figure 5: Result inference choosing the statement 3.65: $p \Rightarrow q \Rightarrow r \equiv p \wedge q \Rightarrow r$ with other fields empty. The symbol *eq* is $\equiv$ because $p \Rightarrow q \Rightarrow r$ and $p \wedge q \Rightarrow r$ are of type $p$.

Leaving only the replacement field empty, the following tree is generated in the backend:

$$\frac{\dfrac{}{\Gamma \vdash [P = Q]}\ \mathrm{P\ id}}{\Gamma \vdash [P[\overline{x} := \overline{Exp}] = Q[\overline{x} := \overline{Exp}]]}\ \mathrm{I}$$

This tree is denoted as $\mathrm{I}^{[\overline{x}:=\overline{Exp}]}\mathrm{P}^{id}$ in applicative notation. It is displayed to the user in the following vertical notation:

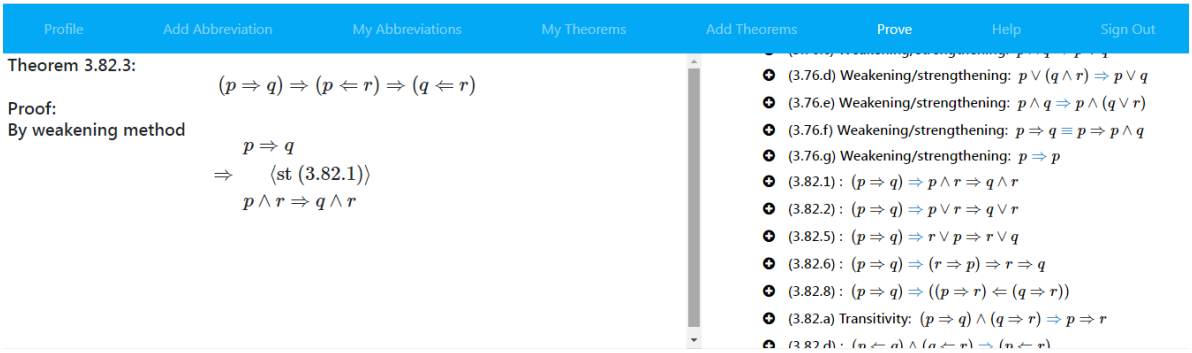



Figure 6: *op*-inference using statement 3.82.1: $(p \Rightarrow q) \Rightarrow p \wedge r \Rightarrow q \wedge r$. The symbol *op* is $\Rightarrow$.

$A_1$
*eq* ⟨st (id) with $\overline{x} := \overline{Exp}$⟩
$A_2$

Where $A_1$ and $A_2$ are the formulas resulting from the calculation of $P[\overline{x} := \overline{Exp}]$ and $Q[\overline{x} := \overline{Exp}]$, respectively.

As before, if $Q$ is `tr` and $P$ is of the form $A_1$ *op* $A_2$, then the display changes in this way:

$A'_1$
*op* ⟨st (id) with $\overline{x} := \overline{Exp}$⟩
$A'_2$

When $A'_1$ and $A'_2$ are the formulas resulting from the calculation of $A_1[\overline{x} := \overline{Exp}]$ and $A_2[\overline{x} := \overline{Exp}]$, respectively (this kind of inference is also called *op*-inference).

Leaving only the substitution field empty, the following tree is generated in the backend:

$$\dfrac{\dfrac{}{\Gamma \vdash [P = Q]}\,\mathrm{P\ id}}{\Gamma \vdash [E^z_P = E^z_Q]}\,\mathrm{R}$$

This tree is denoted as $\mathrm{R}^{\lambda z.E^z}\mathrm{P}^{id}$ in applicative notation. It is displayed to the user in the following vertical notation:

$A_1$
*eq* ⟨st (id) and $E^z : \varphi$⟩
$A_2$

Where $A_1$ y $A_2$ are the formulas resulting from the calculation of $E^z_P$ and $E^z_Q$, respectively.

An example of how CalcLogic displays this type of inference can be found in the Figure 7. Note how, with Rutger Dijkstra's technique, the replacement occurs textually (without renaming the variable $n$).

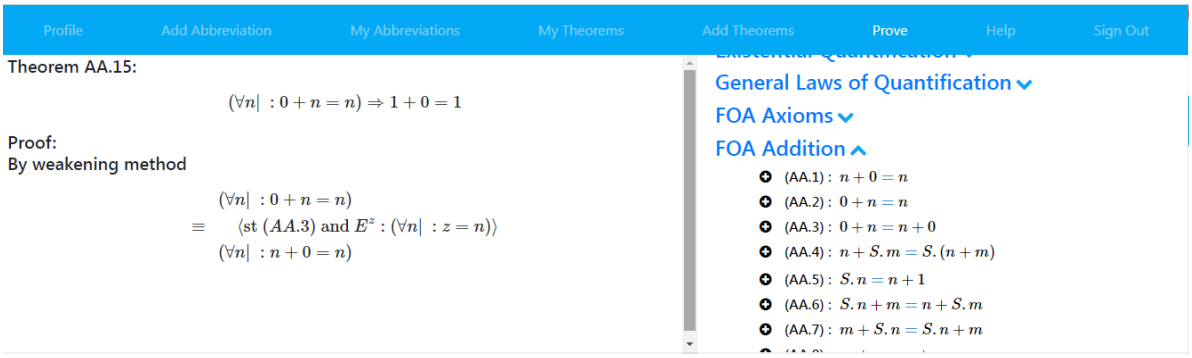



Figure 7: Result of inferring by choosing statement AA.3: $0 + n = n + 0$ filling the replacement field as $(\forall n| : z = n)$.

In some cases, CalcLogic automatically adds a symmetry step such that formulas $A_1$ and $A_2$ swap positions and match the context. For these cases if $\nabla$ is any tree as before with root $\Gamma \vdash [A_1 = A_2]$ and ⟨hint⟩ the respective hint, then the following derivation tree:

$$\dfrac{\nabla}{\Gamma \vdash [A_2 = A_1]}\,\mathrm{S}$$

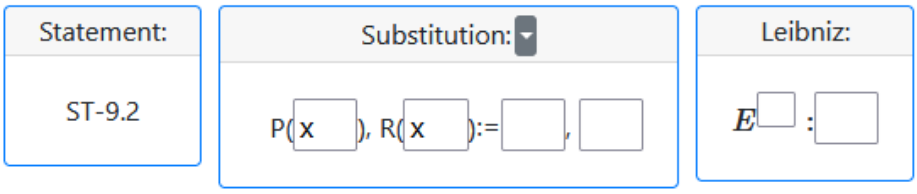



Figure 8: One-step inference form. Including substitution and a field to fill the $E^z$ formula.

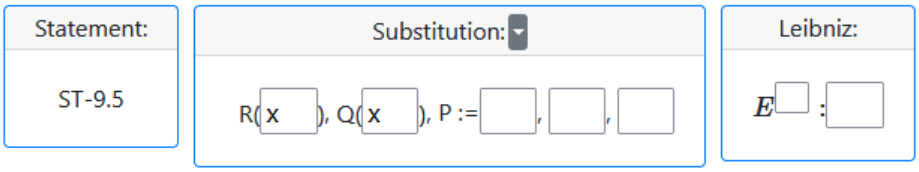



Figure 9: One-step inference form of statement 9.5.

Is written to the user in the following vertical notation:

$$\begin{array}{l} A_2 \\ eq \quad \langle \text{hint} \rangle \\ A_1 \end{array}$$

In this way, the use of the Symmetry rule is transparent to the user.

Although the tree denoted by the above vertical notation is not atomic in general, it is the smallest one-step the user can take. The derivation trees described in this section will be called user's one-step inferences.

In the form of Figure 8 the user selected statement 9.2, which is $(\forall x | R.x : P.x) \equiv (\forall x | : R.x \Rightarrow P.x)$ in CalcLogic. According to a type inference algorithm, the application can detect that the variables $P$ and $R$ are predicates, that is, $P$ and $R$ have types $t \to p$. Since the variables are functions, then in the substitution field of Figure 8 they appear as $P(x), R(x) := \Box, \Box$ where the user can modify $x$ by another letter. As you can see, there is an HTML input to fill the formula corresponding to $P(x)$ and another for $R(x)$. The user can place expressions there. But since $P$ and $Q$ are functions, an abstraction $\lambda x$ is added to the expressions in the back-end. That is, if the user enters $P(x), R(x) := x+2 = 0, true$, CalcLogic processes it as $P, R := \lambda x.x + 2 = 0, \lambda x.true$. That is, $P$ corresponds to the predicate $x + 2 = 0$ dependent on $x$ and $R$ corresponds to the constant predicate $true$ for any value of $x$.

If the user leaves $x$ blank as $P(\Box)$ then the $\lambda$ abstraction is not added to the expression (this is useful when the entered expression is already a function). If the user changes $x$ in $P(x)$ to $P(y)$, then $\lambda y$ is added to the corresponding expression instead of $\lambda x$. This kind of function instantiation does not exist in previous versions of CalcLogic.

### 5.5 Beta Reductions and Quantification

The name of bound variables within a quantifier can be anything. This way, CalcLogic assigns the appropriate name to these variables, preventing them from conflicting with any free variables in the formula. This assignment is a consequence of the $\beta$ reductions. For example, when the user attempts to make a one-step inference by selecting statement 9.5 $P \vee (\forall x | R.x : Q.x) \equiv (\forall x | R.x : P \vee Q.x)$, the forms in Figure 9 appear. The type inference algorithm recognizes that $P$ is a proposition and both $R$ and $Q$ are predicates. Since $P$ is not a function, then by replacing $P$ with a formula in which $x$ occurs, for example $x = x$, it follows that this $x$ is a free variable (CalcLogic does not insert $\lambda x$ into the formula). Therefore, this free variable could conflict with the variable $x$ of the quantifier when instantiating $\mathrm{I}^{[P:=x=x]}\mathrm{P}^{9.5}$.

However, the instantiation rule is derived from the lambda calculus according to section 5.3.6 each time it is invoked. In this case, this derivation is as follows:

$$\dfrac{\dfrac{}{\Gamma \vdash \lambda R, Q, P.P \vee (\Psi \ \wedge \ (\lambda x.R.x)\ (\lambda x.Q.x)) = \lambda R, Q, P.\Psi \ \wedge \ (\lambda x.R.x)\ (\lambda x.P \vee Q.x)}\ \text{P 9.5}}{\Gamma \vdash (\lambda z.\lambda R, Q, x.z\ R\ Q\ (x = x))t_1 = (\lambda z.\lambda R, Q, x.z\ R\ Q\ (x = x))t_2}\ \text{A}$$

Where $t_1$ and $t_2$ abbreviate the left and right sides of statement 9.5 respectively. The beta reductions performed on the right side of the resulting equality are as follows:

$(\lambda z.\lambda R, Q, x.z\ R\ Q\ (x = x))t_2$

$\rightarrow_\beta$
$\lambda R, Q, x.t_2\ R\ Q\ (x = x)$
$=$ $\langle$abbreviation$\rangle$
$\lambda R, Q, x.(\lambda R, Q, P.\Psi\ \wedge\ (\lambda x.R.x)\ (\lambda x.P \vee Q.x))\ R\ Q\ (x = x)$
$\rightarrow_\beta$
$\lambda R, Q, x.(\lambda P.\Psi\ \wedge\ (\lambda x.R.x)\ (\lambda x.P \vee Q.x))\ (x = x)$
$\underset{renaming}{=}$
$\lambda R, Q, x.(\lambda P.\Psi\ \wedge\ (\lambda y.R.y)\ (\lambda y.P \vee Q.y))\ (x = x)$
$\rightarrow_\beta$
$\lambda R, Q, x.\Psi\ \wedge\ (\lambda y.R.y)\ (\lambda y.x = x \vee Q.y)$

Similarly, the $\beta$ normal form of the left side of the root of the previous tree is $\lambda R, Q, x.x = x \vee (\Psi\ \wedge\ (\lambda x.R.x)\ (\lambda x.Q.x))$. In this way, the resulting equality at the root of the tree is

$$\lambda R, Q, x.x = x \vee (\Psi\ \wedge\ (\lambda x.R.x)\ (\lambda x.Q.x)) = \lambda R, Q, x.\Psi\ \wedge\ (\lambda y.R.y)\ (\lambda y.x = x \vee Q.y)$$

which is the representative of the class

$$[x = x \vee (\Psi\ \wedge\ (\lambda x.R.x)\ (\lambda x.Q.x)) = \Psi\ \wedge\ (\lambda y.R.y)\ (\lambda y.x = x \vee Q.y)]$$

The last line encodes the formula $x = x \vee (\forall x|R.x : Q.x) \equiv (\forall y|R.y : x = x \vee Q.y)$, which is the expected result of the substitution. In this way, the $\beta$-reduction algorithm automatically performs all the necessary renaming steps. For example, Figure 10 shows how CalcLogic displays the instantiation of 9.5 with $P := x = x$.

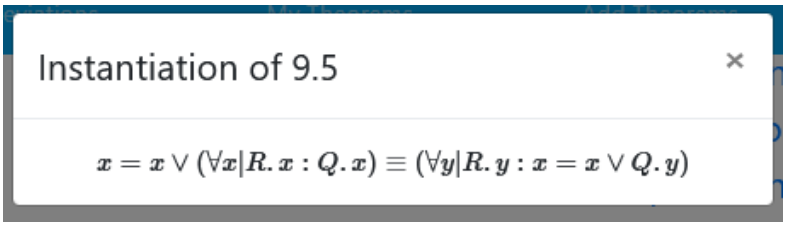



Figure 10: Instantiation of statement 9.5 with $P := x = x$ on CalcLogic.

Statement 9.5 in LADM is stated alongside the condition $\neg occurs('x',' P')$, written in the metalanguage, which requires $x$ to not occur in $P$. This is no longer necessary due to functional coding and automatic renaming. Therefore, a simpler inference system than in LADM has been obtained.

On the other hand, if the substitution field for $Q(x)$ were filled with $x = x$, then, according to the explanation in the previous section, the actual substitution when instantiating the theorem would be $[Q := \lambda x.x = x]$. Since $\lambda x.x = x$ has no free variables, the $\beta$-reductions involving the instantiation $\mathrm{I}^{[Q:=\lambda x.x=x]}\mathrm{P}^{9.5}$ do not rename the variables, and the instantiated formula becomes $P \vee (\forall x|R.x : x = x) \equiv (\forall x|R.x : P \vee x = x)$, as expected.

### 5.6 Metatheorems and Inference Tree Schemes

Metatheorems supported by CalcLogic are those that can be proved using a derivation scheme. Each time a metatheorem is invoked, the application instantiates the scheme by building a specific derivation tree.

#### *5.6.1 Metatheorem 1 $M_1(\nabla)$*

Exercise 3.4 of LADM states as a metatheorem that $P$ is a theorem if and only if $P \equiv true$ is a theorem. (This is the same theorem found in [36] as Theorem 2.2). In our theory this metatheorem works when $[P]$ is of the form $[P = \mathtt{tr}]$, otherwise a combination with the next metatheorem (with *op* as $=_t$ or $\equiv$) can be done to transform the statement into this form. The proof in LADM depends on several theorems. However, in this work the metatheorem is almost a consequence of the foundational principles, in particular the introduction of the internal symbol `tr`.

Therefore, one side of the equivalence of the metatheorem can be proven using the CalcLogic inference rules in the following way:

$$\dfrac{\dfrac{\nabla}{\Gamma \vdash [P = \mathtt{tr}]} \quad \dfrac{\dfrac{}{\Gamma \vdash [true = \mathtt{tr}]}\,\mathrm{P}}{\Gamma \vdash [\mathtt{tr} = true]}\,\mathrm{S}}{\Gamma \vdash [P = true]}\,\mathrm{T}$$

Remember that $[P = true]$ is printed as $P \equiv true$. This derivation requires $[true]$ to be proven in $\Gamma$. The symbol $\nabla$ denotes a tree whose root is $\Gamma \vdash [P = \texttt{tr}]$. To abbreviate this type of derivation scheme, an index will be used to identify the scheme. The index corresponding to this derivation is 1 in our list, so it will be denoted as $M_1(\nabla)$.

From now on, the notation $M_i(\nabla)$ refers to the $i$-th tree scheme where $\nabla$ is a sub-derivation in the scheme. The letter $M$ stands for Metatheorem.

### *5.6.2 Metatheorem 2 $M_2^{op}(\nabla)$*

The following metatheorem is expressed as follows: If *op* is a reflexive operator (i.e. the theorem or axiom $[p\ op\ p]$ exists), and $[P = Q]$ is a theorem, then $[P\ op\ Q]$ is a theorem. The following derivation scheme, denoted as $M_2^{op}(\nabla)$, verifies the validity of the metatheorem:

$$\dfrac{\dfrac{\dfrac{\nabla}{\Gamma \vdash [P = Q]}}{\Gamma \vdash [P\ op\ P = P\ op\ Q]}\,\mathrm{R} \qquad \dfrac{\dfrac{}{\Gamma \vdash [p\ op\ p]}\,\mathrm{P}}{\Gamma \vdash [P\ op\ P]}\,\mathrm{I}}{\Gamma \vdash [P\ op\ Q]}\,\mathrm{E}$$

The statements $[P = Q]$ and $[P =_t Q]$ are not equal, since they refer to $\lambda\overline{x}.P = \lambda\overline{x}.Q$ and $\lambda\overline{x}.(P =_t Q) = \lambda\overline{x}.\texttt{tr}$ respectively. The same can be said when $\equiv$ is used instead of $=_t$. This metatheorem and the next two are used to convert one statement into the other.

### *5.6.3 Metatheorem 3 $M_3(\nabla)$*

The following metatheorem is expressed as follows: If $[P \equiv Q]$ is a theorem, then $[P = Q]$ is a theorem. The following derivation scheme, which is built in four parts, proves the affirmation.

$$\dfrac{\dfrac{\dfrac{\dfrac{\nabla}{\Gamma \vdash [P \equiv Q]}}{\Gamma \vdash [P \equiv Q = true]}\,\mathrm{M_1}(\nabla)}{\Gamma \vdash [(P \equiv Q) \equiv Q = true \equiv Q]}\,\mathrm{R} \qquad \dfrac{\dfrac{}{[true \equiv q = q]}\,\mathrm{P}}{\Gamma \vdash [true \equiv Q = Q]}\,\mathrm{I}}{\Gamma \vdash [(P \equiv Q) \equiv Q = Q]}\,\mathrm{T}$$

If we denote the previous tree as $\nabla'$, then:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [(p \equiv q) \equiv r = p \equiv (q \equiv r)]}\,\mathrm{P}}{\Gamma \vdash [(P \equiv Q) \equiv Q = P \equiv (Q \equiv Q)]}\,\mathrm{I}}{\Gamma \vdash [P \equiv (Q \equiv Q) = (P \equiv Q) \equiv Q]}\,\mathrm{S} \qquad \dfrac{\nabla'}{\Gamma \vdash [(P \equiv Q) \equiv Q = Q]}}{\Gamma \vdash [P \equiv (Q \equiv Q) = Q]}\,\mathrm{T}$$

If we denote the previous tree as $\nabla''$, then:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [true = q \equiv q]}\,\mathrm{P}}{\Gamma \vdash [true = Q \equiv Q]}\,\mathrm{I}}{\Gamma \vdash [P \equiv true = P \equiv (Q \equiv Q)]}\,\mathrm{R} \qquad \dfrac{\nabla''}{\Gamma \vdash [P \equiv (Q \equiv Q) = Q]}}{\Gamma \vdash [P \equiv true = Q]}\,\mathrm{T}$$

If we denote the previous tree as $\nabla'''$, then:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [p \equiv q = q \equiv p]}\,\mathrm{P}}{\Gamma \vdash [P \equiv true = true \equiv P]}\,\mathrm{I} \qquad \dfrac{\dfrac{}{\Gamma \vdash [true \equiv q = q]}\,\mathrm{P}}{\Gamma \vdash [true \equiv P = P]}\,\mathrm{I}}{\Gamma \vdash [P \equiv true = P]}\,\mathrm{T}}{\Gamma \vdash [P = P \equiv true]}\,\mathrm{S} \qquad \dfrac{\nabla'''}{\Gamma \vdash [P \equiv true = Q]}}{\Gamma \vdash [P = Q]}\,\mathrm{T}$$

The tree built combining the previous four trees will be denoted as $\mathrm{M}_3(\nabla)$. Note that $\mathrm{M}_3(\nabla)$ converts a proof of the form $[P \equiv Q]$ into one of the form $[P = Q]$.

To carry out this metatheorem it is necessary that in the premises $\Gamma$ the statements $[(p \equiv q) \equiv r = p \equiv (q \equiv r)]$, $[p \equiv q = q \equiv p]$, $[true]$, $[true \equiv q = q]$ and $[true = q \equiv q]$ are found. All of which are axioms in LADM, except for the last three. The last two are displayed as $(true \equiv q) \equiv q$ and $true \equiv (q \equiv q)$ respectively, and in LADM $true \equiv q \equiv q$ is an axiom, leaving the placement of the parentheses ambiguous. This suggested that one of the statements $[true \equiv q = q]$ and $[true = q \equiv q]$ should be defined as an axiom and the other should be derivable from the associativity of $\equiv$. But because of the internal coding using $=$ it is not possible to use associativity $[(p \equiv q) \equiv r = p \equiv (q \equiv r)]$, to convert $[true \equiv q = q]$ into $[true = q \equiv q]$ or vice versa.

The solution found is to define $[true \equiv q = q]$ and $[q \equiv q]$ as axioms. With these, $[true]$ can be derived by instantiating the two statements with $[q := true]$ and applying Equanimity. Once $[true]$ has been proven, $[true = q \equiv q]$ can be derived using Metatheorem 1 with $[q \equiv q]$ and then symmetry.

$[q \equiv q]$ is not an axiom in LADM, but it is convenient to have it because it allows us to use Metatheorem 2 with $op$ as $\equiv$. Once we have these axioms and theorems, we can use Metatheorems 2 and 3 to derive $[P \equiv Q]$ and $[P = Q]$ from each other. Interderivability allows everything to be done the same as in LADM, since this admits a derivations hidden from the user, to use the associativity of $\equiv$ on statements of the form $P \equiv (Q \equiv R)$ or $(P \equiv Q) \equiv R$, even though their internal encoding is $[P = Q \equiv R]$ or $[P \equiv Q = R]$.

Each of the metatheorems in this work requires that certain premises be found in $\Gamma$. To avoid having to list the statements in $\Gamma$, it is claimed the following: Whenever a statement appears in a derivation tree of this work, by means of the Projection Rule P, this statement is either an axiom or can be proved from the axioms of Propositional and Predicate Logic.

### *5.6.4 Metatheorem 4 $M_4(\nabla)$*

The following metatheorem is expressed as follows: If $[P =_t Q]$ is a theorem, then $[P = Q]$ is a theorem. The metatheorem is proven using the axiom $[e =_t f \Rightarrow E.e \equiv E.f]$ and the following derivation scheme.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [e =_t f \Rightarrow E.e \equiv E.f]}\,\mathrm{P}}{\Gamma \vdash [P =_t Q \Rightarrow E.P \equiv E.Q]}\,\mathrm{I} \quad \dfrac{\nabla}{\Gamma \vdash [P \ =_t \ Q]}}{\Gamma \vdash [E.P \equiv E.Q]}\,\mathrm{MP}}{\Gamma \vdash [E.P = E.Q]}\,M_3}{\Gamma \vdash [P = Q]}\,\mathrm{L}$$

The previous tree will be denoted as $M_4(\nabla)$.

The implementation of the metatheorems is done by means of a string with placeholders that are then replaced and parsed. For example, if $e =_t f \Rightarrow E.r \equiv E.f$ is identified with id, then metatheorem 4 is written in application notation as $\mathrm{L}(\mathrm{M}_3(\mathrm{I}^{[f,e:=\%(Q),\%(P)]}\mathrm{P}^{id}(\%(T))))$. Each time this Metatheorem is called, the placeholders $\%(Q)$, $\%(P)$ and $\%(T)$ are replaced by the specific expressions $Q$, $P$ and the derivation tree $\nabla$. The resulting string is then parsed to obtain the derivation of $[P = Q]$.

### *5.6.5 Metatheorem 5 $M_5(\nabla)$*

There are symbols that are symmetric to some existing ones, for example $\geq$ is a symbol that represents a predicate symmetric to $\leq$ since $x \geq y \equiv y \leq x$ is an axiom. The following metatheorem says that if $[P\ op\ Q]$ is a theorem then $[Q\ op'\ P]$ is a theorem when $op'$ is the symmetric symbol of $op$, that is, when the theorem $[p\ op'\ q = q\ op\ p]$ or $[p\ op\ q = q\ op'\ p]$ exists.

The following derivation tree schemes prove this statement. One tree or the other is used depending on which side $op$ and $op'$ occur in the theorem that defines the symmetry symbol.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [p\ op\ q = q\ op'\ p]}\,\mathrm{P}}{\Gamma \vdash [P\ op\ Q = Q\ op'\ P]}\,\mathrm{I} \quad \dfrac{\nabla}{\Gamma \vdash [P\ op\ Q]}}{\Gamma \vdash [Q\ op'\ P]}\,\mathrm{E}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [p\ op'\ q = q\ op\ p]}\,\mathrm{P}}{\Gamma \vdash [Q\ op'\ P = P\ op\ Q]}\,\mathrm{I}}{\Gamma \vdash [P\ op\ Q = Q\ op'\ P]}\,\mathrm{S} \quad \dfrac{\nabla}{\Gamma \vdash [P\ op\ Q]}}{\Gamma \vdash [Q\ op'\ P]}\,\mathrm{E}$$

The theorems are indexed in a way that facilitates searching for a symbol that had a symmetric symbol. Indexing is done using Broda and Damas combinators [37]; however, the technique is a topic for future work.

### *5.6.6 Metatheorem 6 $M_6^x(\nabla)$*

The statement of this metatheorem is as follows. If $[P]$ is a theorem, $\lambda\overline{y}.P = \lambda\overline{y}.\texttt{tr}$ is its representative, and $x$ is a variable (one of the variables in the list $\overline{y}$ or not), then $[(\forall x| : P)]$ is a theorem. The following tree scheme proves the statement.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\nabla}{\Gamma \vdash [P]}}{\Gamma \vdash [P = true]}\,\mathrm{M}_1}{\Gamma \vdash [(\forall x| : P) = (\forall x| : true)]}\,\mathrm{R} \quad \dfrac{\dfrac{}{\Gamma \vdash [(\forall x|R.x : true) = true]}\,\mathrm{P}}{\Gamma \vdash [(\forall x|true : true) = true]}\,\mathrm{I}}{\Gamma \vdash [(\forall x| : P) = true]}\,\mathrm{T}}{\dfrac{\Gamma \vdash [true = (\forall x| : P)]}{\Gamma \vdash [(\forall x| : P)]}\,\mathrm{S} \qquad \dfrac{}{\Gamma \vdash [true]}\,\mathrm{P}}\,\mathrm{E}$$

The above tree is denoted $\mathrm{M}_6^x(\nabla)$ where $x$ is the bound variable of $\forall x$.

### *5.6.7 Metatheorem 7 $M_7(\nabla)$*

The statement of this metatheorem is as follows. If $[P \wedge Q \Rightarrow R]$ is a theorem, then $[P \Rightarrow Q \Rightarrow R]$ is a theorem. The following tree scheme proofs the statement.

$$\dfrac{\dfrac{\dfrac{\mathrm{P}\ \dfrac{}{\Gamma \vdash [p \wedge q \Rightarrow r = p \Rightarrow q \Rightarrow r]}}{\Gamma \vdash [P \wedge Q \Rightarrow R = P \Rightarrow Q \Rightarrow R]}\,\mathrm{I} \qquad \dfrac{\nabla}{\Gamma \vdash [P \wedge Q \Rightarrow R]}}{\Gamma \vdash [P \Rightarrow Q \Rightarrow R]}\,\mathrm{E}$$

### *5.6.8 Metatheorem 8 $M_8(\nabla)$*

The following metatheorem is expressed as follows: Let be $op$, $op_1$, $op_2$ and $op'$, $op'_1$, $op'_2$ symbols such that $[p\ op\ q = q\ op'\ p]$, $[p\ op_1\ q = q\ op'_1\ p]$, $[p\ op_2\ q = q\ op'_2\ p]$ and $[p\ op_1\ q \wedge q\ op_2\ r \Rightarrow p\ op\ r]$ are axioms or theorems. If $[P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R]$ is a theorem, then $[R\ op'_2\ Q \wedge Q\ op'_1\ P \Rightarrow R\ op'\ P]$ is a theorem. If $\nabla$ is a derivation tree of $[P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R]$, the following derivation scheme, which is built in four parts, proves the affirmation.

$$\dfrac{\mathrm{R}\ \dfrac{\mathrm{P}\ \dfrac{}{\Gamma \vdash [p\ op_1\ q = q\ op'_1\ p]}}{\Gamma \vdash [P\ op_1\ Q = Q\ op'_1\ P]}\,\mathrm{I} \qquad \mathrm{R}\ \dfrac{\mathrm{P}\ \dfrac{}{\Gamma \vdash [p\ op_2\ q = q\ op'_2\ p]}}{\Gamma \vdash [Q\ op_2\ R = R\ op'_2\ Q]}\,\mathrm{I}}{\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R = Q\ op'_1\ P \wedge Q\ op_2\ R \Rightarrow P\ op\ R]}$$

$$\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R = Q\ op'_1\ P \wedge R\ op'_2\ Q \Rightarrow P\ op\ R]$$

If we denote the previous tree as $\nabla'$, then:

$$\dfrac{\dfrac{\nabla'}{\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R = Q\ op'_1\ P \wedge R\ op'_2\ Q \Rightarrow P\ op\ R]} \qquad \mathrm{R}\ \dfrac{\mathrm{P}\ \dfrac{}{\Gamma \vdash [p\ op\ q = q\ op'\ p]}}{\Gamma \vdash [P\ op\ R = R\ op'\ P]}\,\mathrm{I}}{\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R = Q\ op'_1\ P \wedge R\ op'_2\ Q \Rightarrow R\ op'\ P]}$$

$$\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R = Q\ op'_1\ P \wedge R\ op'_2\ Q \Rightarrow R\ op'\ P]$$

If we denote the previous tree as $\nabla''$, then:

$$\dfrac{\dfrac{\nabla''}{\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R = Q\ op'_1\ P \wedge R\ op'_2\ Q \Rightarrow R\ op'\ P]} \qquad \mathrm{R}\ \dfrac{\mathrm{I}\ \dfrac{\mathrm{P}\ \dfrac{}{\Gamma \vdash [p \wedge q = q \wedge p]}}{\Gamma \vdash [Q\ op'_1\ P \wedge R\ op'_2\ Q = R\ op'_2\ Q \wedge Q\ op'_1\ P]}}{\Gamma \vdash [Q\ op'_1\ P \wedge R\ op'_2\ Q \Rightarrow R\ op'\ P = R\ op'_2\ Q \wedge Q\ op'_1\ P \Rightarrow R\ op'\ P]}$$

$$\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R = R\ op'_2\ Q \wedge Q\ op'_1\ P \Rightarrow R\ op'\ P]$$

If we denote the previous tree as $\nabla'''$, then:

$$\dfrac{\dfrac{\nabla'''}{\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R = R\ op'_2\ Q \wedge Q\ op'_1\ P \Rightarrow R\ op'\ P]} \qquad \dfrac{\nabla}{\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R]}}{\Gamma \vdash [R\ op'_2\ Q \wedge Q\ op'_1\ P \Rightarrow R\ op'\ P]}$$

The tree built combining the previous four trees will be denoted as $M_8(\nabla)$. The derivation tree $\nabla$ could be an instantiation of $[p\ op_1\ q \wedge q\ op_2\ r \Rightarrow p\ op\ r]$ or other derivation tree with root $\Gamma \vdash [P\ op_1\ Q \wedge Q\ op_2\ R \Rightarrow P\ op\ R]$.

In case the axioms (or theorems) $[p\ op\ q = q\ op'\ p]$, $[p\ op_1\ q = q\ op'_1\ p]$ or $[p\ op_2\ q = q\ op'_2\ p]$ occur in the database as $[q\ op'\ p = p\ op\ q]$, $[q\ op'_1\ p = p\ op_1\ q]$ or $[q\ op'_2\ p = p\ op_2\ q]$ respectively, then to match the construction above, a symmetry rule S must be used on each occurrence of the axiom in the tree. CalcLogic is able to detect these variants and place the necessary symmetries each time this metatheorem is invoked.

### 5.7 Proof Methods

CalcLogic has two types of proof methods. Recursive proof methods are those that require one or more sub-proofs (each of which has its own proof method). Base proof methods are those that do not require any sub-proofs.

The proof methods are internally identified by tokens like `EO`, `OE`, `DT`, etc., and the way to store the nesting of multiple ones is by using parentheses in an applicative language. For example `MI (AI (CO DT) DT)` is a representation of five nested proof methods.

The application needs information about these proof methods to construct the derivation tree as the user makes sequences of one-step inferences. Additionally, the method is needed to know how to traverse the derivation tree to display it in vertical notation. The derivation trees that CalcLogic constructs in each proof method are discussed in the following sections.

### 5.8 Base Proof Methods

In this section all the base proof methods that CalcLogic supports are explained.

#### *5.8.1 Starting from one side*

From the user's point of view, this method does not change from the previous version of CalcLogic. The only difference is that the method is only used for statements of the form $[P = Q]$ (not of the form $[P =_t Q]$). It consists of using a vertical proof notation starting from $P$ (or $Q$), to arrive at $Q$ (or $P$) in several steps. Within the nested methods notation, this method is denoted as `SL` or `SR` if the proof starts from the left or right, respectively.

For example, if the user selects to start the proof from the left, the first user's one-step inference is $\dfrac{\nabla_1}{\Gamma \vdash [P = A_1]}$ and the second $\dfrac{\nabla_2}{\Gamma \vdash [A_1 = A_2]}$, then the derivations are combined into the following tree.

$$\dfrac{\dfrac{\nabla_1}{\Gamma \vdash [P = A_1]} \quad \dfrac{\nabla_2}{\Gamma \vdash [A_1 = A_2]}}{\Gamma \vdash [P = A_2]}\ \mathrm{T}$$

Then it is printed on the screen to the user in the following vertical notation:

$P$
$eq \quad \langle \mathrm{hint}_1 \rangle$
$A_1$
$eq \quad \langle \mathrm{hint}_2 \rangle$
$A_2$

Where each derivation $\Gamma \vdash A_{i-1}\ eq\ A_i$ ($A_0$ is taken as $P$) is the derivation corresponding to

$A_{i-1}$
$eq \quad \langle \mathrm{hint}_i \rangle$
$A_i$

according to the notation described in Section 5.4.

If the user makes a third inference $\dfrac{\nabla_3}{\Gamma \vdash [A_2 = A_3]}$, then the complete proof takes the form of the following tree:

$$\dfrac{\dfrac{\dfrac{\nabla_1}{\Gamma \vdash [P = A_1]} \quad \dfrac{\nabla_2}{\Gamma \vdash [A_1 = A_2]}}{\Gamma \vdash [P = A_2]}\ \mathrm{T} \quad \dfrac{\nabla_3}{\Gamma \vdash [A_2 = A_3]}}{\Gamma \vdash [P = A_3]}\ \mathrm{T}$$

Then it is printed on the screen to the user in the following vertical notation:

$$\begin{array}{ll} & P \\ eq & \langle \text{hint}_1 \rangle \\ & A_1 \\ eq & \langle \text{hint}_2 \rangle \\ & A_2 \\ eq & \langle \text{hint}_3 \rangle \\ & A_3 \end{array}$$

The proof ends when the term $A_n$ is $Q$. The previous vertical notation tries to tell the user that "$\Gamma \vdash [P = A_1]$ and $\Gamma \vdash [A_1 = A_2]$ and $\Gamma \vdash [A_2 = A_3]$" where "and" is the conjunction of the metalanguage. The intention of this vertical multi-step "$eq \quad \langle \text{hint} \rangle$" notation is to write derivations more compactly than the tree.

Although each user's one-step inference is printed as three lines, when it is printing the entire proof, it is only necessary to print two lines at a time. For example, the vertical notation above is printed from bottom to top, while the printing algorithm traverses the tree in this order: first $\nabla_3$, then $\nabla_2$, and finally $\nabla_1$. First, it is printed $A_3$, then for each $\nabla_i$, it is printed $\overset{A_{i-1}}{eq} \quad \langle \text{hint}_i \rangle$ (only two lines). The process ends when the first line of the vertical notation is printed.

As explained, the last line ($A_3$) of the proof (in vertical notation) is printed first. However, this is not just for algorithmic convenience. The last line is printed using a different algorithm than the others, since certain HTML tags are added to it. These tags make it possible for browser scripts to autocomplete the replacement field when an expression is underlined (see Section 4.2.4 in [4]).

On the other hand, there is another modality of "Starting from one side method". If the user selects to start from the right, then the proof ends when the user reaches $P$. The complete proof is saved as the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\nabla_1}{\Gamma \vdash [Q = A_{n-1}]} \quad \dfrac{\nabla_2}{\Gamma \vdash [A_{n-1} = A_{n-2}]}}{\Gamma \vdash [Q = A_{n-2}]}\,\text{T} \quad \dfrac{\nabla_3}{\Gamma \vdash [A_{n-2} = A_{n-3}]}}{\Gamma \vdash [Q = A_{n-3}]}\,\text{T}}{\begin{array}{c}\vdots \\ \Gamma \vdash [Q = A_1]\end{array} \qquad \dfrac{\nabla_n}{\Gamma \vdash [A_1 = P]}}{\dfrac{\Gamma \vdash [Q = P]}{\Gamma \vdash [P = Q]}\,\text{S}}\,\text{T}$$

Applying the same algorithm to this tree (ignoring the last symmetry rule $S$), the proof is printed to the user vertically as before, but in reverse order (the first line would be $Q$ and the last $P$).

The last symmetry rule in the tree is added when the user performs the last step of the proof. This symmetry is transparent to the user since the printing algorithm ignores it. Most proof methods have to add inferences to the tree to complete the proof. Thus, for each method, CalcLogic has implemented algorithms to: 1) recognize when the user performs the last step. 2) Add inferences to the tree to complete the proof. 3) Print a completed proof ignoring the inferences added in 2).

If the proof method changes, the same vertical notation of this section may represent a tree where the user's one-step inferences $\nabla_i$ are located in different places than in the trees of this section. Therefore, CalcLogic needs the information about the proof method used to interpret a derivation tree and print it correctly.

### *5.8.2 Direct Method*

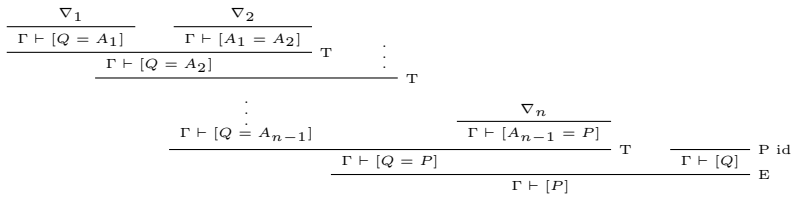
The direct method consists of either using the vertical notation starting from the theorem to be proved and arriving at an already proven theorem, or starting from an already proven theorem and arriving at the one to be proved.

For example, if $P$ is the theorem to prove and $Q$ is a proven theorem with identifier id, then, using the direct method, the following vertical proof

$$\begin{array}{ll} & Q \quad - \ st\ id \\ \equiv & \langle hint_1 \rangle \\ & A_1 \\ \equiv & \langle hint_2 \rangle \\ & A_2 \\ \vdots & \\ \equiv & \langle hint_n \rangle \end{array}$$

$P$

is the user's display of the following derivation tree:



Where each derivation $\Gamma \vdash [A_{i-1} = A_i]$ is the user's one-step inference corresponding to

$$\begin{array}{ll} & A_{i-1} \\ \equiv & \quad \langle hint_i \rangle \\ & A_i \end{array}$$

The proof ends when the user makes the inference that leads to $P$. In that case, CalcLogic automatically adds the equanimity rule to the derivation tree as before.

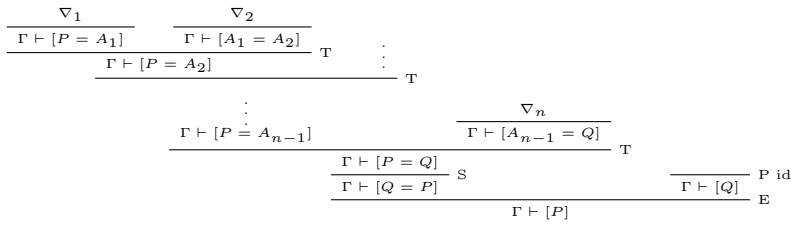
The vertical notation is printed by ignoring the last equanimity rule and then printing the rest of the tree using the same procedure as the previous method. The comment $-$ *st id* next to the formula $Q$ is added at the end of the procedure, where id is obtained from the projection rule to the right of the equanimity.

To be used in the nested methods notation described above, this method is denoted as `DT`.

On the other hand, if $P$ is the theorem to be proved and $Q$ is a proven theorem with identifier id, then, using the direct method, the following vertical proof

$$\begin{array}{ll} & P \\ \equiv & \quad \langle hint_1 \rangle \\ & A_1 \\ \equiv & \quad \langle hint_2 \rangle \\ & A_2 \\ \vdots & \\ \equiv & \quad \langle hint_n \rangle \\ & Q \quad - \ st\ id \end{array}$$

is the user's display of the following derivation tree of $\Gamma \vdash P$



This form of the direct method will be denoted by `DS`.

This proof method was already described in [6], however now there is an additional complication resulting from the new encoding, which forces it to be revised. The statement $[Q]$ can be of the form $[R \equiv S]$ or $[R =_t S]$ while in $\Gamma$ could be in the form $[R = S]$. CalcLogic prints these variants in the same way, and therefore the user has no way to distinguish between them. In this case, CalcLogic automatically executes Metatheorem 2 on the $[R = S]$ at the right of Equanimity to convert it into $[R \equiv S]$ or $[R =_t S]$ depending on the case. In the direct method `DS`, CalcLogic checks whether the last line $Q$ admits equanimity. This search is performed every time the user adds an inference. Using an unification algorithm, CalcLogic can detect whether $Q$ is an instance of a proven theorem or axiom of $\Gamma$.

For example, if 3.1 and 3.2.a identify the statements $[(p \equiv q) \equiv r = p \equiv (q \equiv r)]$ and $[p \equiv q = q \equiv p]$, then the proof:

$$\begin{array}{ll} & P \wedge Q \equiv (q \equiv (q \equiv P \wedge Q)) \\ \equiv & \quad \langle \text{st (3.1) with } p, r := P \wedge Q, q \equiv P \wedge Q \rangle \end{array}$$

$$(P \wedge Q \equiv q) \equiv (q \equiv P \wedge Q) \quad - \text{ st (3.2.a) with } p := P \wedge Q$$

is the user's display of the following derivation tree: $\mathrm{S}(\mathrm{S}(\mathrm{I}^{[p,r:=P\wedge Q,q\equiv P\wedge Q]}\mathrm{P}^{3.1}))\mathrm{M}_2^{\equiv}(\mathrm{I}^{[p:=P\wedge Q]}\mathrm{P}^{3.2.a})$. The tree $\mathrm{S}(\mathrm{I}^{[p,r:=P\wedge Q,q\equiv P\wedge Q]}\mathrm{P}^{3.1})$ is an user's one-step inference, the leftmost S is the fixed symmetry rule needed for `DS` method and $\mathrm{M}_2^{\equiv}(\mathrm{I}^{[p:=P\wedge Q]}\mathrm{P}^{3.2.a})$ is added automatically (the substitution $[p := P \wedge Q]$ is automatically detected). The use of the metatheorem is not printed to the user so that the proof matches the LADM format.

### *5.8.3 Transitivity Method*

In the previous proof methods the user had to do one-steps like $\Gamma \vdash [P = Q]$ with $Q$ different to `tr`, in order to nest the inferences. When there is an axiom or theorem $[p\ op_1\ q \wedge q\ op_2\ r \Rightarrow p\ op\ r]$ (where one or both of the operators $op_1$ and $op_2$ are equal to $op$) the transitivity method allows $op$-inferences. This method is commonly used when $op_1$, $op_2$ and $op$ are any of the relational operators $<$, $>$, $\leq$ and $\geq$. For example, if the statements $[(x+y)^2 = x^2 + 2xy + y^2]$, $[x + 1^2 > x]$, $[x^2 + y \geq y]$, $[x * 1 = x]$ are identified by AM.11, AI.28, AI.29, and AM.5 respectively, then a proof that $[(x+1)^2 > 2x]$ could be:

$$\begin{array}{ll} & (x+1)^2 \\ = & \quad \langle \text{st (AM.11) with } y := 1 \rangle \\ & x^2 + 2x * 1 + 1^2 \\ > & \quad \langle \text{st (AI.28) with } x := x^2 + 2x * 1 \rangle \\ & x^2 + 2x * 1 \\ \geq & \quad \langle \text{st (AI.29) with } y := 2x * 1 \rangle \\ & 2x * 1 \\ = & \quad \langle \text{st (AM.5) with } x := 2x \rangle \\ & 2x \end{array}$$

This method formalizes this type of proof as a derivation tree. The display to the user is in the previous vertical format. If the theorem to be proved is $[P\ op\ Q]$, then the proof starts with the first line $P$ and ends with the last line $Q$. In the nested method notation, this method is encode `TL` which denote that the proof starts on the left ($P$).

The user's one-step inference in this proof method is either $\dfrac{\nabla}{\Gamma \vdash [P\ op\ Q]}$ ($op$-inference) or $\dfrac{\nabla}{\Gamma \vdash [P = Q]}$. The construction of the derivation can be described recursively on the number of $op$-inferences that occur in the tree. The base case is a user's one-step $op$-inference or a tree of the form:

$$\mathrm{M}\,\dfrac{\mathrm{I}\,\dfrac{\mathrm{M}\,\dfrac{\dfrac{}{\Gamma \vdash [e =_t f \Rightarrow E_f^z \Rightarrow E_e^z]}\,\mathrm{P}}{\Gamma \vdash [A_0 =_t A_{k-1} \Rightarrow A_{k-1}\ op\ A_k \Rightarrow A_0\ op\ A_k]} \quad \dfrac{M_2^{=_t}(\nabla')}{\Gamma \vdash [A_0 =_t A_{k-1}]}}{\Gamma \vdash [A_{k-1}\ op\ A_k \Rightarrow A_0\ op\ A_k]} \quad \dfrac{\nabla}{\Gamma \vdash [A_{k-1}\ op\ A_k]}}{\Gamma \vdash [A_0\ op\ A_k]}$$

where $\nabla'$ is a derivation of $\Gamma \vdash [A_0 = A_{k-1}]$ at style Starting From One Side method and $\dfrac{\nabla}{\Gamma \vdash [A_{k-1}\ op\ A_k]}$ is a user's one step $op$-inference. The previous tree is printed to the user like this:

$$\left.\begin{array}{ll} & A_0 \\ = & \langle hint_1 \quad \rangle \\ & \vdots \\ = & \langle hint_{k-1} \rangle \\ & A_{k-1} \\ op & \langle hint_k \rangle \\ & A_k \end{array}\right\} \nabla' \quad \textit{in vertical notation}$$

$$\Big\} \nabla$$

The recursive step of the construction is like this: If $\dfrac{\nabla''}{[A_1\ op_1\ A_k]}$ is a transitivity method derivation tree, we can construct recursively more complex derivation trees in this way:

$$\mathrm{M}\,\dfrac{\mathrm{M}_7\,\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [p\ op_1\ q \wedge q\ op_2\ r \Rightarrow p\ op\ r]}\,\mathrm{P}}{\Gamma \vdash [A_0\ op_1\ A_{k-1} \wedge A_{k-1}\ op_2\ A_k \Rightarrow A_0\ op\ A_k]}\,\mathrm{I}}{\Gamma \vdash [A_0\ op_1\ A_{k-1} \Rightarrow A_{k-1}\ op_2\ A_k \Rightarrow A_0\ op\ A_k]} \quad \dfrac{\nabla''}{\Gamma \vdash [A_0\ op_1\ A_{k-1}]}}{\Gamma \vdash [A_{k-1}\ op_2\ A_k \Rightarrow A_0\ op\ A_k]} \quad \dfrac{\nabla}{\Gamma \vdash [A_{k-1}\ op_2\ A_k]}}{\Gamma \vdash [A_0\ op\ A_k]}$$

or like this:

$$\mathrm{R}\,\frac{\dfrac{\overset{\nabla}{\overline{\Gamma \vdash [A_{k-1} = A_k]}}}{\Gamma \vdash [A_0\ op\ A_{k-1} = A_0\ op\ A_k]} \qquad \dfrac{\nabla''}{\Gamma \vdash [A_0\ op\ A_{k-1}]}}{\Gamma \vdash [A_0\ op\ A_k]}\,\mathrm{E}$$

Where $\overset{\nabla}{\overline{[A_{k-1}\ op_2\ A_k]}}$ and $\overset{\nabla}{\overline{[A_{k-1} = A_k]}}$ are user's one-step inferences. The first tree is printed to the user in this way:

$$\left.\begin{array}{l}\vdots\\ op_2\end{array}\right\}\nabla''\ \textit{in vertical notation}$$
$$\left.\begin{array}{c}A_{k-1}\\ \langle hint\rangle\\ A_k\end{array}\right\}\nabla$$

and the second one is printed in this way:

$$\left.\begin{array}{l}\vdots\\ =\end{array}\right\}\nabla''\ \textit{in vertical notation}$$
$$\left.\begin{array}{c}A_{k-1}\\ \langle hint\rangle\\ A_k\end{array}\right\}\nabla$$

The printing algorithm is the same as before, except that all user's one-step inferences are searched in different places in the tree. If the entire tree is not a single user's one-step inference, the search is done as follows: 1) If the root of the tree is obtained by transitivity rule, the search is performed as in Starting From One Side. 2) If the tree ends with modus ponens, the last user's one-step inference $\nabla$ is in the right child of the root. 3) If the tree ends with equanimity, the search of $\nabla$ is performed above the replacement rule of the left child of the root. Then, the search is continued recursively in $\nabla''$ (for cases 2 and 3).

On the other hand, this proof method can automatically use Metatheorem 5 when there is an operator $op'$ symmetric to $op$. For example, if the user tries to perform the $op$-inference $\overset{\nabla}{\overline{\Gamma \vdash [A_k\ op'\ A_{k-1}]}}$ in the $k$-th step, CalcLogic detects that it should invert the root of this step to $[A_{k-1}\ op\ A_k]$. To do this, it automatically applies Metatheorem 5.

In the `TL` method, add hidden inferences to complete the proof is not necessary after the last step. With these constructions the root of the tree will be the statement to be proved after the last user's one-step inference. This will not be true for the other kind of this method explained below.

When there exists an operator $op'$ symmetric to $op$, the proof of $[P\ op\ Q]$ can be started from the right ($Q$) using the $op'$ symbol instead of $op$ in the steps. This kind of transitivity method is encoded with the token `TR` to denote that it starts on the right. In this case the proof ends when the last line is $P$ (which is when the root of the tree is $\Gamma \vdash [Q\ op'\ P]$). Upon reaching $P$ using the `TR` method, CalcLogic adds (hidden to the user) a Metatheorem 5 to the entire tree to end the proof.

For example, if the identification of the statements $[e =_t f \Rightarrow E_f^z \Rightarrow E_e^z]$ and $[p > q \wedge q \geq r \Rightarrow p > r]$ are 3.83.b and AI.10 respectively then, the derivation tree corresponding to the proof of $[(x+1)^2 > 2x]$ made above, using the `TL` method, is:

$(\mathrm{R}^{\lambda z.(x+1)^2 > z}(\mathrm{I}^{[x:=2x]}\mathrm{P}^{\mathrm{AM}.5}))$
$((\mathrm{M}_7(\mathrm{I}^{[p,q,r:=(x+1)^2,x^2+2x*1,2x*1]}\mathrm{P}^{\mathrm{AI}.10}))$
$(\mathrm{I}^{[e,f,E:=(x+1)^2,x^2+2x*1+1,\lambda z.z>x^2+2x*1]}\mathrm{P}^{3.83.b}(\mathrm{M}_2^{=_t}(\mathrm{I}^{[y:=1]}\mathrm{P}^{\mathrm{AM}.11}))(\mathrm{I}^{[x:=x^2+2x*1]}\mathrm{P}^{\mathrm{AI}.28}))$
$(\mathrm{I}^{[y:=2x*1]}\mathrm{P}^{\mathrm{AI}.29})$
$)$

However, the same proof using the `TR` method requires exchanging the equalities with S rule, inequalities with Metatheorem 5 and the transitivity statement AI.10 with the Metatheorem 8 as follows.

$(\mathrm{R}^{\lambda z.2x<z}(S(\mathrm{I}^{[y:=1]}\mathrm{P}^{\mathrm{AM}.11})))$
$((\mathrm{M}_7(\mathrm{M}_8(\mathrm{I}^{[p,q,r:=(x+1)^2,x^2+2x*1,2x*1]}\mathrm{P}^{\mathrm{AI}.10})))$
$(\mathrm{I}^{[e,f,E:=2x,2x*1,\lambda z.z\geq x^2+2x*1]}\mathrm{P}^{3.83.b}(\mathrm{M}_2^{=_t}(S(\mathrm{I}^{[y:=2x]}\mathrm{P}^{\mathrm{AM}.5})))(\mathrm{M}_5(\mathrm{I}^{[y:=2x*1]}\mathrm{P}^{\mathrm{AI}.29})))$
$(\mathrm{M}_5(\mathrm{I}^{[x:=x^2+2x*1]}\mathrm{P}^{\mathrm{AI}.28}))$
$)$

Considering that the identifiers of the statements $[p \geq q = q \leq p]$ and $[p > q = q < p]$ are AI.12 AI.13 respectively, then the above proof is printed to the user as

$$\begin{array}{l}2x\\ =\quad \langle \text{st (AM.5) with } x := 2x\rangle\\ 2x*1\end{array}$$

$\leq$ $\langle$st (AI.29, AI.12) with $y := 2x * 1\rangle$
$x^2 + 2x * 1$
$<$ $\langle$st (AI.28, AI.13) with $x := x^2 + 2x * 1\rangle$
$x^2 + 2x * 1 + 1^2$
$=$ $\langle$st (AM.11) with $y := 1\rangle$
$(x + 1)^2$

(Note: The above derivation tree is rooted at $\Gamma \vdash [2x < (x + 1)^2]$. To obtain the derivation of the initial theorem $\Gamma \vdash [(x + 1)^2 > 2x]$, CalcLogic adds an application of Metatheorem 5 to the whole tree, but this is not printed to the user.)

#### *5.8.4 Weakening-Strengthening Method*

This method is the same as the transitivity method but where $op$, $op_1$ and $op_2$ are all equal to $\Rightarrow$ or all equal to $\Leftarrow$. When the statement to be proved is $[P \Rightarrow Q]$ or $[P \Leftarrow Q]$ the method is called weakening or strengthening if the proof starts from the antecedent or consequent respectively.

There are two differences with respect to the previous method, one hidden and the other visible to the user: The first is that in the first tree described in the previous method, the premise $[e =_t f \Rightarrow E^z_f \Rightarrow E^z_e]$ must be changed to $[e \equiv f \Rightarrow E^z_f \Rightarrow E^z_e]$ and the metatheorem $M_2^{=_t}(\nabla')$ to $M_2^{\equiv}(\nabla')$ (this avoids a type error). The second is a functionality that is explained below.

The user only can use the Replacement rule to statements of the form $[P = Q]$ with $Q$ different to `tr`. However this method allows inferences in which some implication is used in a sub-formula. In those cases the same notation $E$ as Replacement is used for de hint. For example if formula $p \Rightarrow p \vee q$ has identifier 2 then we can write

$p \wedge q \Rightarrow p$
$\Rightarrow$ $\langle$st 2 and $E^z : p \wedge q \Rightarrow z\rangle$
$p \wedge q \Rightarrow p \vee q$

However in [1] it is explained that depending on the place of a subformula, denoted by $z$ in $E^z$, the implication can change the direction. For example if $z$ is located in the antecedent of an implication the inference looks like this:

$p \wedge q \Rightarrow p$
$\Leftarrow$ $\langle$st 2 and $E^z : z \wedge q \Rightarrow p\rangle$
$(p \vee q) \wedge q \Rightarrow p$

According to the terminology used in [1], formula scheme $E^z : p \wedge q \Rightarrow z$ defines a monotonic predicate transformer, while formula scheme $E^z : z \wedge q \Rightarrow p$ defines an anti monotonic one.

The last two vertical notation represent a derivation tree $\nabla$ with root $(p \wedge q \Rightarrow p) \Rightarrow (p \wedge q \Rightarrow p \vee q)$ or $(p \wedge q \Rightarrow p) \Leftarrow ((p \vee q) \wedge q \Rightarrow p)$ respectively. The construction of $\nabla$ is by means of an algorithm that is described in Section 5.10.1. The rule that specifies the direction of the main logic connector ($\Rightarrow$ or $\Leftarrow$) depending of the position of $z$ is a statement called metatheorem of Parity [1]. Our algorithm is another way to find the right logic connector ($\Rightarrow$ or $\Leftarrow$), therefore the rule described in the metatheorem of parity according [1] can be ignored in this work.

### 5.9 Recursive Proof Methods

As previously stated, recursive proof methods generate one or more sub-proofs. Those methods are further categorized into linear and branching methods. Linear recursive proof methods generate a single sub-proof, while branching recursive proof methods generate two sub-proofs. Only the linear ones will be discussed in this section.

Most linear recursive methods to prove $[Q]$ consist of proving $[P]$, since $P$ and $Q$ are equivalent. Therefore, a method of this form changes the statement to be proven, and this proof becomes a sub-proof from the user's perspective.

Once the user constructs a derivation tree $\nabla$ to prove $[P]$, the proof ends up using the following tree scheme.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [p = q]}\,\mathrm{P}}{\Gamma \vdash [P = Q]}\,\mathrm{I} \qquad \dfrac{\nabla}{\Gamma \vdash [P]}}{\Gamma \vdash [Q]}\,\mathrm{E}$$

In the case where the premise in $\Gamma$ is $[q = p]$ instead of $[p = q]$, the last tree is modified as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [q = p]}\,\mathrm{P}}{\Gamma \vdash [Q = P]}\,\mathrm{I}}{\Gamma \vdash [P = Q]}\,\mathrm{S} \qquad \dfrac{\nabla}{\Gamma \vdash [P]}}{\Gamma \vdash [Q]}\,\mathrm{E}$$

It is called contrapositive (`CR`), mutual implication (`MI`) or contradiction (`CO`) when $[p = q]$ in the previous trees is $[p \Rightarrow q = \neg q \Rightarrow \neg p]$, $[(p \Rightarrow q) \wedge (q \Rightarrow p) = p \equiv q]$, or $[\neg p \Rightarrow false = p]$, respectively. On the other hand, there is a method that allows the user to choose the statement $[p = q]$ to use, it is called the Equanimity Method. Once the statement $[p = q]$ is chosen, CalcLogic is able to identify whether the first or second tree should be used depending on what is to be proved. Equanimity method is encoded with the token `EL` if the first tree is used or `ER` if the second tree is used.

When the entire proof is printed with a recursive method, CalcLogic only prints the subtree $\nabla$, but with a comment at the beginning and another at the end of the proof. The initial and final comments depend on the method chosen. For example, in Figure 11, it can be seen that the initial comment of a proof using Mutual Implication is the declaration of the new statement that the user must prove.

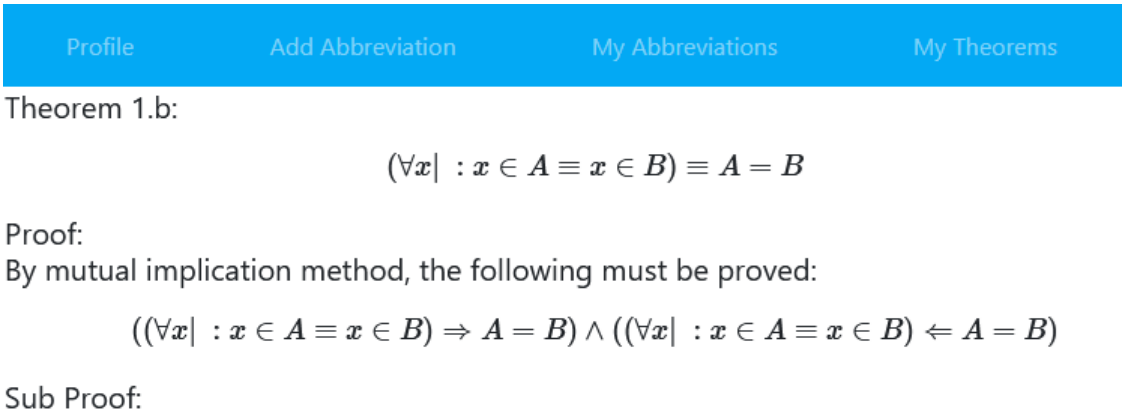



Figure 11: Proof of a theorem beginning with Mutual Implication method

The rest of the linear recursive methods have derivation schemes to finish the proof that differ from the previous methods. The following are examples of this:

### *5.9.1 Equality to Operator*

A theorem of the form $[R = S]$ (where $S$ is not `tr`) cannot be proved by direct method, because just before the final equanimity the derivation must be $\Gamma \vdash [Q = (R = S)]$ with $Q \in \Gamma$, but this is impossible because nesting function equality $=$ is not allowed. However, CalcLogic prints this statement as $R\ op\ S$ where $op$ is $\equiv$ if $R$ and $S$ are of type $p$ or $=_t$ if they are of type $t$, so the user cannot know the true form of the statement. In this case, this method is hidden from the user, who changes the formula to be proved to $[R\ op\ S]$, which is the one proved by the direct method. After finishing the proof of $[R\ op\ S]$, the method ends by applying Metatheorem 3 to the entire derivation tree if $op$ is $\equiv$, or Metatheorem 4 if it is $=_t$. If $op$ is $\equiv$, this method is denoted as `EO` otherwise as `EE`.

### *5.9.2 Operator to Equality*

$[P =_t Q]$ cannot be proved by starting from one side; it has to be in the form $[P = Q]$. The user cannot distinguish between the two statements because they are printed the same. Thus, if you try to prove $[P =_t Q]$ by starting from one side, CalcLogic hides this recursive method that converts the formula to be proved into $[P = Q]$. After proving $[P = Q]$ by starting from one side, the proof ends by applying Metatheorem 2 to the entire derivation tree $\nabla$ (taking $op$ to be $=_t$ in $M_2^{op}(\nabla)$). This method is denoted `OE`.

### *5.9.3 Modus Ponens Method*

It is used when you want to prove $[Q]$ but an implication $[p \Rightarrow q]$ has already been proved that can be instantiated in $[P \Rightarrow Q]$. When selecting this method together with the theorem $[p \Rightarrow q]$, CalcLogic asks the user to prove $[P]$ instead of $[Q]$. Once $[P]$ is proved, the following tree is built to complete the proof:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [p \Rightarrow q]}\,\mathrm{P}}{\Gamma \vdash [P \Rightarrow Q]}\,\mathrm{I} \qquad \dfrac{\nabla}{\Gamma \vdash [P]}}{\Gamma \vdash [Q]}\,\mathrm{M}$$

The instantiation of the selected implication $[p \Rightarrow q]$ is done automatically to fit the context of the theorem. For example, Figure 12 shows a proof (in Arithmetic theory) with the nesting of two modus ponens methods, the first selecting $[p \Rightarrow q]$ as the statement 9.13: $[(\forall x| : P.x) \Rightarrow P.e]$ and the second as the induction axiom FOA.3: $[P.0 \wedge (\forall n| : P.n \Rightarrow P.(S.n)) \Rightarrow (\forall n| : P.n)]$ (The symbol $S$ is the successor function). Note that CalcLogic automatically detects the substitution for the predicate $P$ so that it fits the context. The token used to denote this method is `MP`, therefore these nested methods are saved in the database as the string `MP MP`. When the proof is loaded, this string is parsed and the application detects the nesting, deduces the substitutions and prints the initial part of the proof in Figure 12.

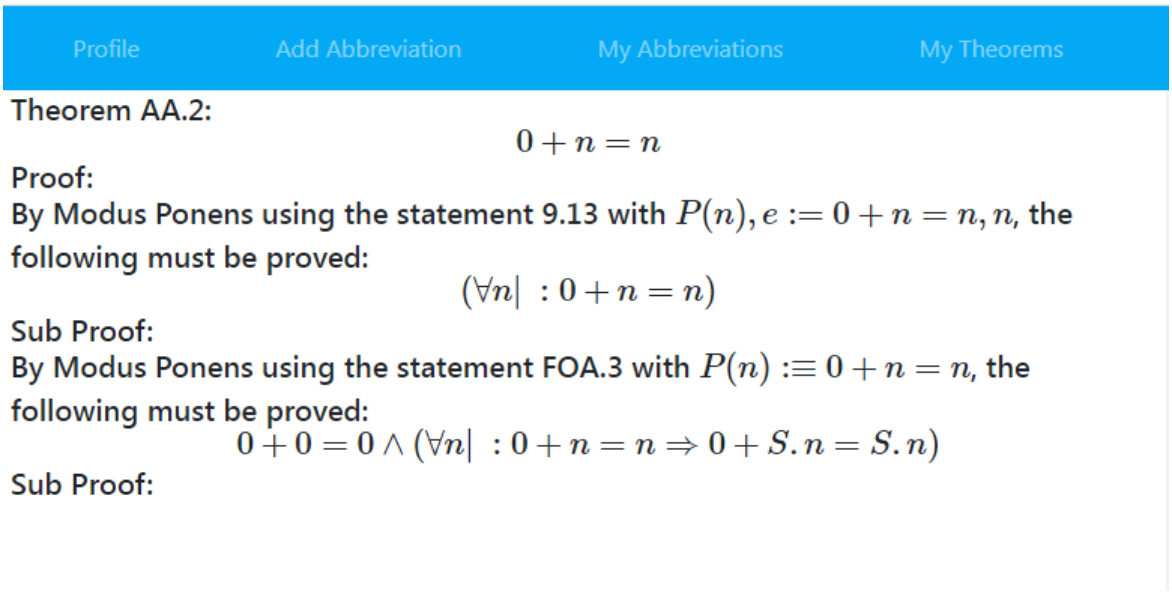



Figure 12: Nested Modus Ponens methods using first 9.13:$(\forall x| : P.x) \Rightarrow P.e$ and then FOA.3: $P.0 \wedge (\forall n| : P.n \Rightarrow P.(S.n)) \Rightarrow (\forall n| : P.n)$ with $P(n) := (0 + n = n)$.

Note that in Arithmetic theory, this chaining of methods generates a conjunction that contains the base case and inductive case of a proof by induction.

### *5.9.4 Generalization*

This method is used to prove statements of the form $[(\forall x|R.x : P.x)]$, by simply proving $[R.x \Rightarrow P.x]$ for an arbitrary $x$. Once $[R.x \Rightarrow P.x]$ is proved, the following tree is built to complete the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [(\forall x|r.x : p.x) = (\forall x| : r.x \Rightarrow p.x)]}\,\mathrm{P}}{\Gamma \vdash [(\forall x|R.x : P.x) = (\forall x| : R.x \Rightarrow P.x)]}\,\mathrm{I}}{\Gamma \vdash [(\forall x| : R.x \Rightarrow P.x) = (\forall x|R.x : P.x)]}\,\mathrm{S} \qquad \dfrac{\dfrac{\nabla}{\Gamma \vdash [R.x \Rightarrow P.x]}}{\Gamma \vdash [(\forall x| : R.x \Rightarrow P.x)]}\,\mathrm{M}_6(\nabla)}{\Gamma \vdash [(\forall x|R.x : P.x)]}\,\mathrm{E}$$

The Generalization method can also be used to prove $[(\forall x| : P.x)]$ by proving $[P.x]$. In that case, just add Metatheorem 6 to the derivation of $[P.x]$ to complete the final proof.

For example, in Figure 13 shows the proof of the inductive case of the last subproof in Figure 12. The proof is made by generalization and then weakening. The token used to denote this method is `GE`.

### *5.9.5 Witness*

This method is used to prove $(\exists x| : P.x) \Rightarrow Q$ by simply proving $P.x \Rightarrow Q$ ($x$ must not occur in $Q$). Once the user proves $P.x \Rightarrow Q$, the following tree is built to complete the proof:

Statement:
$$(\forall n|\ : 0+n=n \Rightarrow 0+S.n=S.n)$$
Sub Proof:
By generalization method, the following must be proved considering an arbitrary n:
$$0+n=n \Rightarrow 0+S.n=S.n$$
Sub Proof:
By weakening method
$$\begin{array}{ll} & 0+n=n \\ \Rightarrow & \langle \text{st } (3.83.d) \text{ with } E,f,e := S.x,n,0+n\rangle \\ & S.(0+n)=S.n \\ \equiv & \langle \text{st } (AA.4) \text{ with } m,n := n,0 \text{ and } E^z : z=S.n\rangle \\ & 0+S.n=S.n \end{array}$$
since n was arbitrary, then the statement can be generalized
$$\therefore\ (\forall n|\ : 0+n=n \Rightarrow 0+S.n=S.n)$$

Figure 13: Proof by Generalization. The initial comment of the proof says that $n$ is arbitrary and the closing comment says that $n$ was arbitrary and therefore can be generalized.

$$\dfrac{\dfrac{\mathrm{P}\ \dfrac{}{\Gamma \vdash [(\forall x| : p.x \Rightarrow q) = (\exists x| : p.x) \Rightarrow q]}}{\mathrm{I}\ \dfrac{}{\Gamma \vdash [(\forall x| : P.x \Rightarrow Q) = (\exists x| : P.x) \Rightarrow Q]}} \qquad \dfrac{\dfrac{\nabla}{\Gamma \vdash [P.x \Rightarrow Q]}}{\Gamma \vdash [(\forall x| : P.x \Rightarrow Q)]}\ \mathrm{M}_6(\nabla)}{\Gamma \vdash [(\exists x| : P.x) \Rightarrow Q]}\ \mathrm{E}$$

This method also allows us to prove $(\exists x|R.x : P.x) \Rightarrow Q$ by proving $R.x \wedge P.x \Rightarrow Q$. To finish the proof in this case, the same tree as above is built (denoted $\nabla'$) and the following is added:

$$\dfrac{\dfrac{\dfrac{\mathrm{P}\ \dfrac{}{\Gamma \vdash [(\exists x|r.x : p.x) = (\exists x| : r.x \wedge p.x)]}}{\mathrm{I}\ \dfrac{}{\Gamma \vdash [(\exists x|R.x : P.x) = (\exists x| : R.x \wedge P.x)]}}}{\mathrm{R}\ \dfrac{}{\Gamma \vdash [(\exists x|R.x : P.x) \Rightarrow Q = (\exists x| : R.x \wedge P.x) \Rightarrow Q]}} \qquad \dfrac{\nabla'}{\Gamma \vdash [(\exists x| : R.x \wedge P.x) \Rightarrow Q]}}{\mathrm{S}\ \dfrac{}{\Gamma \vdash [(\exists x| : R.x \wedge P.x) \Rightarrow Q = (\exists x|R.x : P.x) \Rightarrow Q]}} \qquad \dfrac{}{\Gamma \vdash [(\exists x|R.x : P.x) \Rightarrow Q]}\ \mathrm{E}$$

When CalcLogic asks the user to prove $P.x \Rightarrow Q$, it automatically renames $x$ if it occurs in $Q$. This ensures that the quantified variable does not occur in $Q$. This renaming is achieved in the same way as in Section 5.5. The token used to denote this method is `WI`.

### *5.9.6 Proof by Parts*

The method is used to prove theorems of the form $[T_1 \wedge T_2]$, by independently proving first $T_1$ and then $T_2$. This is the only branching recursive proof method, in the sense that it allows two subproofs. Each of these can be branched into two more subproofs if the subproof statement has the form required for this method. A piecewise proof is displayed with the following vertical notation:

Statement:
$T_1$
Subproof:
Proof of $T_1$
Statement:
$T_2$
Subproof:
Proof of $T_2$
Using And Introduction:
$\therefore T_1 \wedge T_2$

Where "Proof of $T_1$" and "Proof of $T_2$" are the vertical notation abbreviations for the derivation trees $\dfrac{\nabla_1}{\Gamma \vdash [T_1]}$ and $\dfrac{\nabla_2}{\Gamma \vdash [T_2]}$. The above proof is an abbreviation for the following tree:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash [p = p \wedge true]}\,\mathrm{P}}{\Gamma \vdash [T_1 = T_1 \wedge true]}\,\mathrm{I} \qquad \mathrm{R}\,\dfrac{\mathrm{S}\,\dfrac{\dfrac{\nabla_2}{\Gamma \vdash [T_2]}\,\mathrm{M}_1(\nabla_2)}{\Gamma \vdash [T_2 = true]}}{\Gamma \vdash [true = T_2]}}{\Gamma \vdash [T_1 = T_1 \wedge T_2]}\,\mathrm{T} \qquad \dfrac{\nabla_1}{\Gamma \vdash [T_1]}}{\Gamma \vdash [T_1 \wedge T_2]}\,\mathrm{E}$$

CalcLogic knows the location of $\nabla_1$ and $\nabla_2$ in the above tree, so it can search for them and print them recursively in vertical notation according to the above format.

While the user only made the first subproof, only the tree $\nabla_1$ is saved in the database. When the second subproof starts, the previous tree is built and saved. $\nabla_1$ is the completed first subproof, and $\nabla_2$ is what is carried over into the second subproof. For each user step in the second subproof, $\nabla_2$ is updated, and the previous tree is built again. The proof ends when the root of the tree is the statement to be proven.

The token used to denote this method is `AI`. CalcLogic can tell whether the user is proving the first or second subproof by reading the nested methods notation. For example, `AI SL` states that the first subproof is being done by starting from one side method. Conversely, `AI SL GE` indicates that the second subproof started with generalization method, but still needs a base proof method. Finally, `AI SL (GE WE)` states that the second subproof is done by generalization and then weakening method.

For example, the subproof in the last line of Figure 12 can be done in parts. This last line is a conjunction of two formulas: the base and inductive cases of a natural induction proof. Using Proof by Parts, the base case would be one subproof and the inductive case another subproof. If the base case is proved with `SL` and the inductive step with `GE WE` as in Figure 13, then the nested method string would be `MP (MP (AI SL (GE WE)))`. This string is stored in the database and used to determine which part of the derivation tree corresponds to each subproof. It is also used to determine the printing algorithm to use depending on the methods for each subproof.

### 5.10 Special Metatheorems

There are two metatheorems involving the construction of derivation trees that cannot be described as a schema in the style of section 5.6. The first is the parity metatheorem, and the second is the deduction metatheorem. The second states that if there exists a derivation $\Gamma \cup \{H\} \vdash P$ where $H$ has never been instantiated, then there exists a derivation $\Gamma \vdash H \Rightarrow P$. This latter metatheorem justifies the method of assuming $H$ as a hypothesis to prove $H \Rightarrow P$. However, the proof of deduction metatheorem is complicated and will be left for future work.

#### *5.10.1 Parity Metatheorem*

According to Section 5.8.4 some logical connectors are monotonic with respect to one of their arguments and anti monotonic with respect to others. For this reason, depending on where the $z$ is in Replacement expression $E$, when an *op* inference is made, it can be of the form $\overset{P}{\Rightarrow}\langle\text{hint}\rangle\ q$ or $\overset{P}{\Leftarrow}\langle\text{hint}\rangle\ q$ regardless of whether *op* is $\Rightarrow$ or $\Leftarrow$. However, CalcLogic calculates and displays automatically the correct choice.

This subsection explains the algorithm that the application uses to build the correct derivation tree. This algorithm had been presented in [6], but since in that work there was no instantiation inference rule nor the $\Psi$ operator, then it is necessary to update the algorithm here.

To present the algorithm, the following trees are defined, where $op_1$ is any of the operators $\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$. It is taken that $op \in \{\Rightarrow, \Leftarrow\}$ and $op'$ is $\Leftarrow$ if $op$ is $\Rightarrow$ or $op'$ is $\Rightarrow$ if $op$ is $\Leftarrow$:

$WSL_1^{P,Q,op,R,op_1}$ with $op_1$ any operator between $\wedge$, $\vee$ and $\Leftarrow$

$$\dfrac{\dfrac{}{\Gamma \vdash (p\ op\ q) \Rightarrow ((p\ op_1\ r)\ op\ (q\ op_1\ r))}\,\text{Ax id}}{\Gamma \vdash (P\ op\ Q) \Rightarrow ((P\ op_1\ R)\ op\ (Q\ op_1\ R))}\,\mathrm{I}$$

$WSR_1^{P,Q,op,R,op_1}$ with $op_1$ any operator between $\wedge$, $\vee$ and $\Rightarrow$

$$\dfrac{\dfrac{}{\Gamma \vdash (p\ op\ q) \Rightarrow ((r\ op_1\ p)\ op\ (r\ op_1\ q))}\,\text{Ax id}}{\Gamma \vdash (P\ op\ Q) \Rightarrow ((R\ op_1\ P)\ op\ (R\ op_1\ Q))}\,\mathrm{I}$$

$WSR_2^{P,Q,op,R}$

$$\dfrac{\dfrac{}{\Gamma \vdash (p\ op\ q) \Rightarrow ((r \Leftarrow p)\ op'\ (r \Leftarrow q))}\ \text{Ax id}}{\Gamma \vdash (P\ op\ Q) \Rightarrow ((R \Leftarrow P)\ op'\ (R \Leftarrow Q))}\ \text{I}$$

$WSL_2^{P,Q,op,R}$

$$\dfrac{\dfrac{}{\Gamma \vdash (p\ op\ q) \Rightarrow (p \Rightarrow r\ op'\ q \Rightarrow r)}\ \text{Ax id}}{\Gamma \vdash (P\ op\ Q) \Rightarrow (P \Rightarrow R\ op'\ Q \Rightarrow R)}\ \text{I}$$

$Neg^{P,Q,op}$

$$\dfrac{\dfrac{}{\Gamma \vdash (p\ op\ q) \equiv (\neg p\ op'\ \neg q)}\ \text{Ax id}}{\Gamma \vdash (P\ op\ Q) \equiv (\neg P\ op'\ \neg Q)}\ \text{I}$$

$WSU_1^{\lambda x.P,\lambda x.Q,op,\lambda x.R}$

$$\dfrac{\dfrac{}{\Gamma \vdash (\forall x| : p.x\ op\ q.x) \Rightarrow (\forall x|r.x : p.x)\ op\ (\forall x|r.x : q.x)}\ \text{Ax id}}{\Gamma \vdash (\forall x| : P\ op\ Q) \Rightarrow (\forall x|R : P)\ op\ (\forall x|R : Q)}\ \text{I}$$

$WSU_2^{\lambda x.P,\lambda x.Q,op,\lambda x.R}$

$$\dfrac{\dfrac{}{\Gamma \vdash (\forall x| : p.x\ op\ q.x) \Rightarrow (\forall x|p.x : r.x)\ op'\ (\forall x|q.x : r.x)}\ \text{Ax id}}{\Gamma \vdash (\forall x| : P\ op\ Q) \Rightarrow (\forall x|P : R)\ op'\ (\forall x|Q : R)}\ \text{I}$$

$WSE_1^{\lambda x.P,\lambda x.Q,op,\lambda x.R}$

$$\dfrac{\dfrac{}{\Gamma \vdash (\forall x| : p.x\ op\ q.x) \Rightarrow (\exists x|r.x : p.x)\ op\ (\exists x|r.x : q.x)}\ \text{Ax id}}{\Gamma \vdash (\forall x| : P\ op\ Q) \Rightarrow (\exists x|R : P)\ op\ (\exists x|R : Q)}\ \text{I}$$

$WSE_2^{\lambda x.P,\lambda x.Q,op,\lambda x.R}$

$$\dfrac{\dfrac{}{\Gamma \vdash (\forall x| : p.x\ op\ q.x) \Rightarrow (\exists x|p.x : r.x)\ op\ (\exists x|q.x : r.x)}\ \text{Ax id}}{\Gamma \vdash (\forall x| : P\ op\ Q) \Rightarrow (\exists x|P : R)\ op\ (\exists x|Q : R)}\ \text{I}$$

If $\nabla$ is a derivation where the root is of the form $P\ op\ Q$, then instead of placing the arguments $P, Q, op$ in the abbreviations of the above trees, we can place $\nabla$ directly to simplify the notation. For example $WSL_1^{\nabla,R,op_1}$, $WSR_1^{\nabla,R,op_1}$, $WSR_2^{\nabla,R}$, $WSL_2^{\nabla,R}$ and $Neg^{\nabla}$ will denote $WSL_1^{P,Q,op,R,op_1}$, $WSR_1^{P,Q,op,R,op_1}$, $WSR_2^{P,Q,op,R}$, $WSL_2^{P,Q,op,R}$ and $Neg^{P,Q,op}$, respectively. On the other hand, we will use the notation $\lambda x.\nabla$ in the sense that $WSU_1^{\lambda x.\nabla,\lambda x.R}$, $WSU_2^{\lambda x.\nabla,\lambda x.R}$, $WSE_1^{\lambda x.\nabla,\lambda x.R}$ and $WSE_2^{\lambda x.\nabla,\lambda x.R}$ will denote $WSU_1^{\lambda x.P,\lambda x.Q,op,\lambda x.R}$, $WSU_2^{\lambda x.P,\lambda x.Q,op,\lambda x.R}$, $WSE_1^{\lambda x.P,\lambda x.Q,op,\lambda x.R}$ and $WSE_2^{\lambda x.P,\lambda x.Q,op,\lambda x.R}$ respectively.

It is denoted as $[z]$ the bracket abstraction to abstract the variable $z$ in applicative languages, which in [38] is denoted as $[z]_{BD_{-\eta}}$ (Definition 3.10 of [38]). The bracket abstraction $[z]$ implements the operator $\lambda z$ within combinatory logic. That is, given an expression $Exp$, the result of $[z]Exp$ is an anonymous function that depends on $z$. This function is implemented with combinators and the functional application with terms rewriting.

The combinators used for $[z]$ are those of [37]. These combinators are functions whose purpose will be explained in the paragraph below. The combinators of [37] are syntactically of the form $\Phi_\alpha$, where $\alpha$ is a list of symbols. These symbols are $b$, $c$ and ordered pairs $(\alpha_1, \alpha_2)$ with $\alpha_1$, $\alpha_2$ lists built recursively in the same way that $\alpha$. The definition of $[z]$ is the following:

- $[z]p = \Phi$ if $z = p$,
- $[z]p = \Phi_K p$ if $z \notin p$
- $[z]pq = t_1(q, [z]p)$ if $z \in p \wedge z \notin q$,
- $[z]pq = t_2(p, [z]q)$ if $z \notin p \wedge z \in q$,

- $[z]pq = t_3([z]p, [z]q)$ if $z \in p \wedge z \in q$.

where

$$t_1(q, r_1) = \begin{cases} \Phi_{c\alpha} q w_1 \dots w_{\#\alpha} & if \quad r_1 = \Phi_\alpha w_1 \dots w_{\#\alpha} \\ \Phi & otherwise \end{cases}$$

$$t_2(p, r_2) = \begin{cases} \Phi_{b\alpha} p w_1 \dots w_{\#\alpha} & if \quad r_2 = \Phi_\alpha w_1 \dots w_{\#\alpha} \\ \Phi & otherwise \end{cases}$$

$$t_3(r_1, r_2) = \begin{cases} \Phi_{(\alpha_1,\alpha_2)} w_1 \dots w_{\#\alpha_1} w'_1 \dots w'_{\#\alpha_2} & if \quad r_1 = \Phi_{\alpha_1} w_1 \dots w_{\#\alpha_1} \text{ and } r_2 = \Phi_{\alpha_2} w'_1 \dots w'_{\#\alpha_2} \\ \Phi & otherwise \end{cases}$$

The previous bracket abstraction $[z]$ is the same as in [37], but without the recursive rule $\eta$ and defined with Bunder's algorithm [39], which is more efficient than [37]. The $[z]$ operator converts an expression or formula into a function, for example applying $[z]$ to the formula $\wedge\ z\ \mathbf{T}$ results in $[z](\wedge\ z\ \mathbf{T}) = t_1(\mathbf{T}, [z](\wedge\ z)) = t_1(\mathbf{T}, t_2(\wedge, [z]z)) = t_1(\mathbf{T}, t_2(\wedge, \Phi)) = t_1(\mathbf{T}, \Phi_b\ \wedge) = \Phi_{cb}\ \mathbf{T}\ \wedge$, where the combinator $\Phi_{cb}$ represents the function $\lambda x, y, z.y\ z\ x$. In Johnsson's technique [40], the action of converting the formula $\wedge\ z\ \mathbf{T}$ into the functional version $(\lambda x, y, z.y\ z\ x)\ \ \mathbf{T}\ \wedge$ is called Lifting and the function $(\lambda x, y, z.y\ z\ x)$ Lifter. The bracket abstraction $[z]$ of [37] applied to an expression or formula $Exp$, performs a lambda lifting but in combinatory logic, i.e. $[z]Exp = \Phi_\alpha t_1 \cdots t_n$ where $t_1, \dots, t_n$ are subterms of $Exp$ and $\Phi_\alpha$ is a Lifter (but in combinatory logic).

The term rewriting system that describes the behavior of the functional application to a combinator $\Phi_\alpha$, is defined in [37]. With this nomenclature the following theorem determines the algorithm used to implement the parity metatheorem.

**Theorem 1.** *Let op be $\Rightarrow$ or $\Leftarrow$, let $\nabla$ be a derivation tree of $\Gamma \vdash P\ op\ Q$ and a WFP $E$ of type $p$, with $z$ a variable of type $p$ or $t \to p$ occurring in $E$ only once, without it being in the scope of a $\equiv$'s arguments. Let $E'$ be the same phrase as $E$ but without the lambda abstractions that reach the variable $z$. Let $f$ be a function that receives a term from the applied Lambda Calculus, a derivation tree $\nabla$ and returns another derivation tree according to the recursive rules written below. Then $f([z]E', \nabla)$ is always a tree whose root is $\Gamma \vdash E^z_P\ op_2\ E^z_Q$ or $\Gamma \vdash E^z_{(\lambda x.P)}\ op_2\ E^z_{(\lambda x.Q)}$ (for some $x$) if $z$ is of type $p$ or $t \to p$ respectively, the symbol $op_2$ is some operator between $\Rightarrow$ and $\Leftarrow$. The definition of $f$ is as follows:*

$f(t, \nabla) = \nabla$ *if $t$ is atomic*
$f(t\ \neg, \nabla) = f(t, Neg^\nabla \nabla)$
$f(t\ (\Rightarrow R), \nabla) = f(t, WSL_2^{\nabla,R} \nabla)$
$f(t\ (o\ R), \nabla) = f(t, WSL_1^{\nabla,R,o} \nabla)$
$f(t\ ((\Psi\ \wedge)\ (\lambda x.R')), \nabla) = f(t, WSU_1^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla))$
$f(t\ ((\Psi\ \vee)\ (\lambda x.R')), \nabla) = f(t, WSE_1^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla))$
$f((t\ R) \Leftarrow, \nabla) = f(t, WSR_2^{\nabla,R} \nabla)$
$f((t\ R)\ o, \nabla) = f(t, WSR_1^{\nabla,R,o} \nabla)$
$f((t\ (\lambda x.R'))(\Psi\wedge), \nabla) = f(t, WSU_2^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla))$
$f((t\ (\lambda x.R'))(\Psi\vee), \nabla) = f(t, WSE_2^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla))$
*where $t$ and $R$ are formulas written in applicative language and $o \in \{\Rightarrow, \Leftarrow, \wedge, \vee\}$*

In the proof of Theorem 1, the notation $|p|q$ from Definition 2.4 of [38] is used, which is defined recursively as $|p|(qt) := |pq|t$, $|p|q := pq$ and $|p|\epsilon := p$. Also the notation $z \in E$ will denote "$z$ occurs in $E$".

*Proof.* If $E$ is an atom then $E = z$ because $z \in E$ and $z$ must be of type $p$, in this case $t = \Phi_\epsilon = [z]z$ and therefore $f(t, \nabla) = \nabla$, so that the root of $f(t, \nabla)$ is $P\ op\ Q = E^z_P\ op\ E^z_Q$.

On the other hand, if $E \neq z$, since $z$ is of type $p$ or $t \to p$ then the smallest well-formed phrase $E$ of type $p$ where $z$ occurs only once without being under the scope of a $\equiv$'s arguments, are the following: $\neg z$, $o\ R\ z$ and $o\ z\ R$ where $o \in \{\Rightarrow, \Leftarrow, \wedge, \vee, \Psi\ \wedge, \Psi\ \vee\}$ and $R$ is a WFP. Let's split by cases.

- If $E = \neg z = E'$:

  $t = [z]E = \Phi_b \neg$ so that $f(t, \nabla) = f(\Phi_b, Neg^\nabla \nabla) = Neg^\nabla \nabla$ and this derivation by equanimity has root $\neg P\ op'\ \neg Q = E^z_P\ op'\ E^z_Q$

- If $E = o\ R\ z = E'$ let's split by cases: $o \in \{\Rightarrow\}$ or $o \in \{\Leftarrow, \wedge, \vee\}$ or $o \in \{(\Psi\ \wedge)\}$ or $o \in \{(\Psi\ \vee)\}$.

  - If $o$ is $\Rightarrow$ then $t = [z](\Rightarrow R\ z) = \Phi_b(\Rightarrow R)$, so that $f(t, \nabla) = f(\Phi_b, WSL_2^{\nabla,R} \nabla) = WSL_2^{\nabla,R} \nabla$ and this derivation by modus ponens has root $(P \Rightarrow R)\ op'\ (Q \Rightarrow R)$, that in applicative notation is equals to $(\Rightarrow R\ P)\ op'\ (\Rightarrow R\ Q) = (\Rightarrow R\ z)^z_P\ op'\ (\Rightarrow R\ z)^z_Q = E^z_P\ op'\ E^z_Q$

- If $o \in \{\Leftarrow, \wedge, \vee\}$ then $t = [z](o\ R\ z) = \Phi_b(o\ R)$, so that $f(t, \nabla) = f(\Phi_b, WSL_1^{\nabla,R,o}\nabla) = WSL_1^{\nabla,R,o}\nabla$ and this derivation by modus ponens has root $(P\ o\ R)\ op\ (Q\ o\ R)$, that in applicative notation is equals to $(o\ R\ P)\ op\ (o\ R\ Q) = E_P^z\ op\ E_Q^z$
- If $o \in \{\Psi\ \wedge\}$ then $t = [z]((\Psi\ \wedge)\ R\ z) = \Phi_b((\Psi\ \wedge)\ R)$, $R$ must be of the form $\lambda x.R'$ so that $f(t, \nabla) = f(\Phi_b, WSU_1^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla)) = WSU_1^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla)$ and this derivation by modus ponens has root $(\forall x|R' : P)\ op\ (\forall x|R' : Q)$, that in applicative notation is equals to $(\Psi\ \wedge\ (\lambda x.R')\ (\lambda x.P))\ op\ (\Psi\ \wedge\ (\lambda x.R')\ (\lambda x.Q)) = E_{\lambda x.P}^z\ op\ E_{\lambda x.Q}^z$
- If $o \in \{\Psi\ \vee\}$ then $t = [z]((\Psi\ \vee)\ R\ z) = \Phi_b((\Psi\ \vee)\ R)$, $R$ must be of the form $\lambda x.R'$ so that $f(t, \nabla) = f(\Phi_b, WSE_1^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla)) = WSE_1^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla)$ and this derivation by modus ponens has root $(\exists x|R' : P)\ op\ (\exists x|R' : Q)$, that in applicative notation is equals to $(\Psi\ \vee\ (\lambda x.R')\ (\lambda x.P))\ op\ (\Psi\ \vee\ (\lambda x.R')\ (\lambda x.Q)) = E_{\lambda x.P}^z\ op\ E_{\lambda x.Q}^z$

- If $E = o\ z\ R = E'$ let's split by cases: $o \in \{\Leftarrow\}$ or $o \in \{\Rightarrow, \wedge, \vee\}$ or $o \in \{(\Psi\ \wedge)\}$ or $o \in \{(\Psi\ \vee)\}$.
  - If $o$ is $\Leftarrow$ then $t = [z](\Leftarrow z\ R) = \Phi_{cb}R \Leftarrow$, so that $f(t, \nabla) = f(\Phi_{cb}, WSR_2^{\nabla,R}\nabla) = WSR_2^{\nabla,R}\nabla$, and this derivation by modus ponens has root $(R \Leftarrow P)\ op'\ (R \Leftarrow Q)$ that in applicative notation is equals to $(\Leftarrow P\ R)\ op'\ (\Leftarrow Q\ R) = E_P^z\ op'\ E_Q^z$.
  - If $o \in \{\Rightarrow, \wedge, \vee\}$ then $t = [z](o\ z\ R) = \Phi_{cb}R\ o$, so that $f(t, \nabla) = f(\Phi_{cb}, WSR_1^{\nabla,R,o}\nabla) = WSR_1^{\nabla,R,o}\nabla$, and this derivation by modus ponens has root $(R\ o\ P)\ op\ (R\ o\ Q)$ that in applicative notation is equals to $(o\ P\ R)\ op\ (o\ Q\ R) = E_P^z\ op\ E_Q^z$
  - If $o \in \{\Psi\ \wedge\}$ then $t = [z]((\Psi\ \wedge)\ z\ R) = \Phi_{cb}R\ (\Psi\ \wedge)$, $R$ must be of the form $\lambda x.R'$ so that $f(t, \nabla) = f(\Phi_{cb}, WSU_2^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla)) = WSU_2^{\lambda x.\nabla, \lambda x.R'} M_6^x(\nabla)$ and this derivation by modus ponens has root $(\forall x|P : R')\ op'\ (\forall x|Q : R')$, that in applicative notation is equals to $(\Psi\ \wedge\ (\lambda x.P)\ (\lambda x.R'))\ op'\ (\Psi\ \wedge\ (\lambda x.Q)\ (\lambda x.R')) = E_{\lambda x.P}^z\ op'\ E_{\lambda x.Q}^z$
  - If $o \in \{\Psi\ \vee\}$ then $t = [z]((\Psi\ \vee)\ z\ R) = \Phi_{cb}R\ (\Psi\ \vee)$, $R$ must be of the form $\lambda x.R'$ so that $f(t, \nabla) = f(\Phi_{cb}, WSE_2^{\lambda x.\nabla, \lambda R'} M_6^x(\nabla)) = WSE_2^{\lambda x.\nabla, \lambda R'} M_6^x(\nabla)$ and this derivation by modus ponens has root $(\exists x|P : R)\ op\ (\exists x|Q : R)$, that in applicative notation is equals to $(\Psi\ \vee\ (\lambda x.P)\ (\lambda x.R'))\ op\ (\Psi\ \vee\ (\lambda x.Q)\ (\lambda x.R')) = E_{\lambda x.P}^z\ op\ E_{\lambda x.Q}^z$

Any other term $E \neq z$ can be decomposed in the form $E = (t_1)_{t_2}^z$ with $t_2$ any of the forms $\neg z$, $o\ R\ z$ and $o\ z\ R$ and $z$ occurring in $t_1$ only once. Therefore $E' = t_1'[z := t_2]$ where $t_1'$ is as $t_1$ but without the lambda abstractions that reach the variable $z$. The proof is concluded by doing structural induction on $t_1$.

- If $t_1$ is atomic:

  Then $t_1 = z = t_1'$ because $z \in t_1$, therefore $E' = t_2$ and the proof consists of the verification made previously.

- If $t_1$ is not atomic:

  Is easy to proof that $t = [z]E' = ||\Phi_{\alpha\alpha'}|w|w'$ where $[z]t_1' = |\Phi_\alpha|w$ and $[z]t_2 = |\Phi_{\alpha'}|w'$, where $[z]t_2$ is one of the expressions $t$ previously calculated, for which it has already been shown that $f$ constructs a derivation $\nabla'$ such that the root is $(t_2)_{P'}^z\ op_2\ (t_2)_{Q'}^z$ where $P'$, $Q'$ could be $P$, $Q$ or $\lambda x.P$, $\lambda x.Q$. In this way $f(t, \nabla) = f(||\Phi_{\alpha\alpha'}|w|w', \nabla) = f(|\Phi_{\alpha\alpha'}|w, \nabla') \overset{f\ not\ depend\ of\ \Phi_{\alpha\alpha'}}{=} f(|\Phi_\alpha|w, \nabla')$ and the latter by inductive hypothesis is a derivation whose root is $(t_1)_{(t_2)_{P'}^z}^z\ op_2\ (t_1)_{(t_2)_{Q'}^z}^z = ((t_1)_{t_2}^z)_{P'}^z\ op_2\ ((t_1)_{t_2}^z)_{Q'}^z = E_{P'}^z\ op_2\ E_{Q'}^z$.

□

For example, let the statement $[p \Rightarrow p \vee q]$ in $\Gamma$ be identified as 3.76.a, which is instantiated such that the op-inference $\nabla$ has root $[Q.y \Rightarrow Q.y \vee S]$. Let $E$ be the following $(\exists x|R.x : (\forall y|z : P.x.y))$, which in applicative language would be $\Psi \vee\ (\lambda x.R\ x)\ (\lambda x.\Psi \wedge\ z\ (\lambda y.P\ x\ y))$. Thus $E'$ is $\Psi \vee\ (\lambda x.R\ x)\ (\Psi \wedge\ z\ (\lambda y.P\ x\ y))$ and

$f([z]E', \nabla) = f(\Phi_{bcb}(\Psi \vee\ (\lambda x.R\ x))(\lambda y.P\ x\ y)(\Psi\wedge), \nabla) =$
$WSE_1^{\lambda x.\Psi\wedge(\lambda y.Q\ y)(\lambda y.P\ x\ y), \lambda x.\Psi\wedge(\lambda y.(Q\ y)\vee S)(\lambda y.P\ x\ y), \Leftarrow, \lambda x.R\ x}$
$M_6^x(WSU_2^{\lambda y.Q\ y, \lambda y.(Q\ y)\vee S, \Rightarrow, \lambda y.P\ x\ y} M_6^y(\nabla))$.

The above inference tree has root $[(\exists x|R.x : (\forall y|Q.y : P.x.y)) \Leftarrow (\exists x|R.x : (\forall y|R.y \vee S : P.x.y))]$ and is interpreted as a single inference step that is printed to the user as in Figure 14.

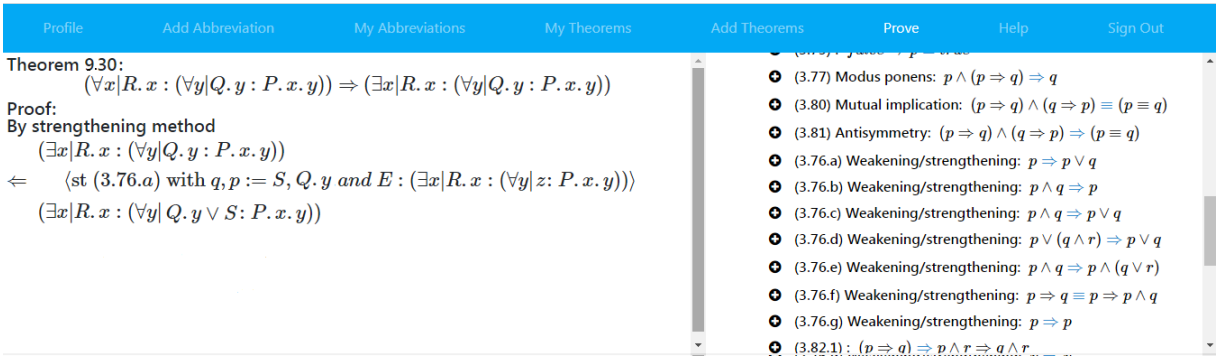



Figure 14: Metatheorem of parity apply to the one-step op-inference using the statement $p \Rightarrow p \vee q$ with the substitution $q, p := S, Q.y$ on the context $E^z : (\exists x|R.x : (\forall y|z : P.x.y))$

## 6 Experiment

This section explains and shows the results of an educational experiment to validate the ease of use of the interface and to justify the use of CalcLogic in learning Symbolic Logic.

### 6.1 Experiment Description

The educational research was divided into two parts: A group of 34 students who had prior knowledge of Calculational Logic and had already been assessed. A second group of 32 students who had partial knowledge of Calculational Logic and were missing assessments.

The surveys conducted with both groups were done in order to answer the research questions: 1) Does the user interface and error messages of CalcLogic enable the student to improve their skills in correctly proving theorems of Propositional Logic and Predicate Logic? 2) Could a user with knowledge of Calculational Logic be able to recognize how to use the application?.

Below are the questions that were asked to the students when the Symbolic Logic course finished.

#### *6.1.1 Questions about the tool*

1. How do you evaluate the CalcLogic tool?
2. What was your impression of using the tool?
3. What was the level of complexity of the tool?

#### *6.1.2 Questions about knowledge of Calculational Logic for using CalcLogic*

4. Do you think your prior knowledge of Calculational Logic helped you understand and use CalcLogic?
5. Did you use your prior knowledge of Calculational Logic in solving exercises within the tool?
6. Do you believe that having used CalcLogic beforehand would have facilitated your understanding of Calculational Logic?

#### *6.1.3 Questions about the contribution of CalcLogic*

7. What was your level of knowledge of Calculational Logic before using CalcLogic?
8. What is your level of knowledge of Calculational Logic after using CalcLogic?

#### *6.1.4 Questions about Competencies and grades*

- Did the use of CalcLogic help you understand concepts or processes of Calculational Logic?
- Which concept or process of Calculational Logic do you consider CalcLogic helped you better understand?
- What was your final grade in the Symbolic Logic course?

### 6.2 Results

In Group I, there were initially 33 students, eight (8) withdrew, and of those who remained enrolled, only 32% responded to the survey. In Group II, of the 22 students enrolled, 4 withdrew and 31% responded to the survey.

Table 1 contains the results corresponding to Part 6.1.1. These results encourage us to continue with the approach adopted to develop the tool's interface, as these suggest that it is easy to use.

Table 1: Results of Questions about the tool: Q1 to Q3

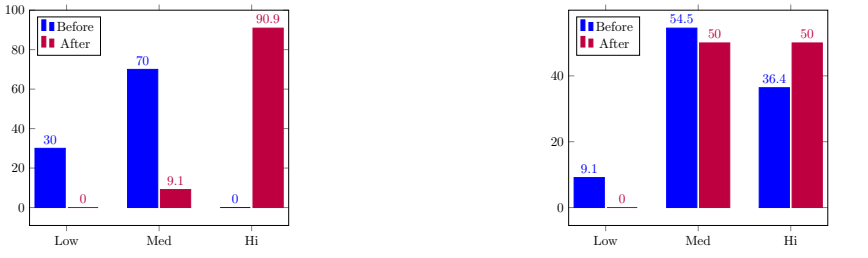
| | Questions about the tool | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Q1 | | | Q2 | | | Q3 | | |
| | Bad | Good | Exc | Hard | Med | Easy | Hi | Med | Low |
| Group I (%) | 0 | 45 | 54 | 0 | 54.5 | 45.5 | 0 | 54.5 | 45.5 |
| Grooup II (%) | 0 | 60 | 40 | 0 | 40 | 60 | 0 | 60 | 40 |

In Table 2, the results of Part 6.1.2 are shown, both groups considered that having prior knowledge of calculational logic allowed them to understand the use of the tool and its application in solving exercises. This is even considering that group 2 had less knowledge of Calculational Logic than group 1, which suggests that not much knowledge is needed to learn how to use the tool. On the other hand, 50% of group 2 and 90.9% of group 1 believed that having used the application before would have facilitated their understanding of calculational logic. Since group 2 started with less competencies, this result suggests that although not much competencies are necessary, a certain minimum is adequate to start using the application.

Table 2: Results of Questions about knowledge of Calculational Logic for using CalcLogic: Q4 to Q6

| | Questions about knowledge of equational logic for using CalcLogic | | | | | |
|---|---|---|---|---|---|---|
| | Q4 | | Q5 | | Q6 | |
| | Yes | No | Yes | No | Yes | No |
| Group I (%) | 90.9 | 9.1 | 100 | 0 | 90.9 | 9.1 |
| Grooup II (%) | 90 | 10 | 100 | 0 | 50 | 50 |

Figure 15 shows the analysis of the questions corresponding to Section 6.1.3. This is an indication that the application helps to gain competencies on the subject. Regarding Group I, it can be observed that there was a significant change of the knowledge according to the survey. A change in knowledge is also reported in group 2 but less abruptly.




(a) Group I (b) Group II

Figure 15: Comparative of level of knowledge before and after using CalcLogic

To analyze the contribution of the tool, corresponding to section 6.1.4, the results are plotted in Figure 16 for each group. The majority of both groups considered that the use of the tool helped them understand concepts and processes in Logic.

The student final qualification in the course and the student pass rate are shown on Figure 17.

The Table 3 and 4 show the results of using the application and the final grade of the course. Of a total of 43 students (in both courses), the average of the last two written assessments (out of four) and final scores were taken by groups, depending on how much their CalcLogic score was.

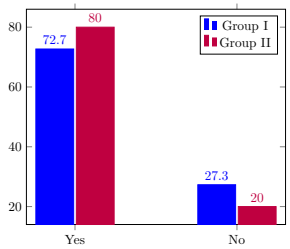
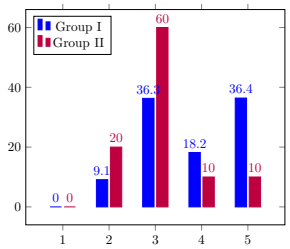
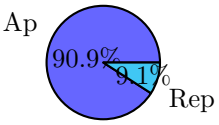


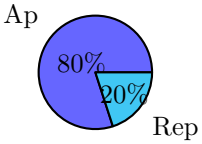

Figure 16: Contribution of CalcLogic




(a) Final Qualification. From 1 to 5 points




(b) % of approbation of Group I




(c) % of approbation of Group II

Figure 17: Final qualifications and percentage of approbation

Table 3 contains the scores of the students of Group I such that: only opened CalcLogic (1 points), tried to prove (2 to 4 points), solved some exercises (5 to 7 points), performed well (greater than 8 points). These results are in Table 3. The use of the application had a weighting of 12.5 points.

Table 3: Average and Final score for Group I

| Group I | CalcLogic score | Average Score (last 2 exams) | Final Score |
|---|---|---|---|
| open | $x = 1$ | 1 / 20 % | 4% |
| try | $2 \leq x < 5$ | 2 / 20 % | 8% |
| solve a few | $5 \leq x < 8$ | 4 / 20 % | 16% |
| good performance | $x \geq 8$ | 18 / 20 % | 72% |

Similarly to the table above, Table 4 shows the average for Group II. The use of the application had a weighting of 10 points for Group II.

Table 4: Average and Final score for Group II

| Group II | CalcLogic score | Average Score (last 2 exams) | Final Score |
|---|---|---|---|
| open | $x = 1$ | 2 / 20 % | 11.11% |
| try | $2 \leq x < 5$ | 0 / 20 % | 0% |
| solve a few | $5 \leq x < 7$ | 4 / 20 % | 22.22% |
| good performance | $x \geq 8$ | 12 /20 % | 66.67% |

The plots in Figure 18 show linear regressions that, for the small sample used, reveal an apparent correlation between the average grade obtained in CalcLogic assessment and the average of the last two

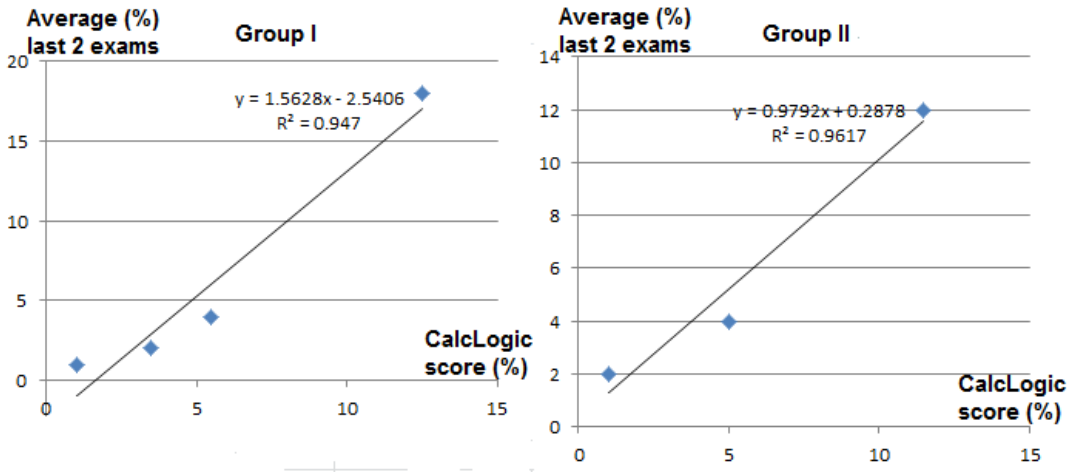



Figure 18: Linear correlations last 2 exams vs CalcLogic score separated into groups according to the tables 3 and 4

written exams. Both groups show this trend.

Table 5: Comparative

| | Group I | Group II |
|---|---|---|
| Number of Students | 25 | 18 |
| CalcLogic (average) Qualification | 11.6 / 12.5 | 7.8 / 10 |
| Final Qualification | 58.72 / 100 | 56.21/ 100 |

Table 5 has a comparative between the scores of both groups based on the CalcLogic qualifications and the Final qualifications. Group I used CalcLogic after finishing the assessments and Group II used the tool before taking the last two (out of four) exams. It is noteworthy that the weighting in CalcLogic was higher in Group I, 2.5 points more than Group II, which affected the final grade; the average final grade of Group I was 2.5 points higher than the average of Group II.

## 7 Conclusions

Russell and Whitehead's Principia Mathematica [41] is an example of how complicated the use of a strict formal system can be, since it is known that it took them 749 pages to prove $1 + 1 = 2$. In our experience, all works that contain mathematical proofs using a strict formal system are difficult to read. This can be observed throughout the 20th century in the works on logic and arithmetic by Curry [9], Church [8], Harold Simmons [33], etc. However, LADM is a book that makes compact formal proofs that even inexperienced students can process, covering certain contents of discrete mathematics in just a few pages. This is the greatest merit of the book, in our opinion.

The notational miracle of LADM is due to the Dijkstra-Scholten system itself, since it is a schematic logic that admits certain shortcuts. In fact, Gries in [42] shows how to make a compact proof of the theorem called Andrew's Challenge, which is one of the benchmarks used to measure the simplicity of proofs of a formal system.

Loading all the theorems of known mathematics into a proof assistant is a daunting task. According to Terence Tao [43], we will see a growth in communities of mathematicians dedicated to this goal. An assistant that automates calculational logic can help in this task, as proofs are often produced faster than with other formal systems. This paper presents initial ideas for implementing a web-based assistant with these features. The medium-term goal is for communities of mathematicians to be able to collaborate on loading theorems from anywhere using the web application.

Regarding the educational experiment: It suggest that there is a correlation between the grades of the assessments made with CalcLogic and the written exams. This had also been observed in [5].

The results seems to show that the tool helps students acquire competencies in the subject matter, even if they are already familiar with it. In fact, those who acquired the most competencies in this experiment were those who were already familiar with the subject matter. The information extracted from the data seems to show that the answer to the first research question is affirmative.

The evidence suggests that the application should not be used in the early stages of the course, preferably after taking the first exams.

On the other hand, it was qualitatively observed that those familiar with Calculational Logic quickly adapted to the application's interface. Both groups proved all the theorems assigned as exercises in just a few days (100 theorems approximately), and they were never given an instruction manual. For this reason, we conclude that surely the answer to the second research question is affirmative (even though the experiment was done with few students).

No information could be extracted from the data to show that any of the answer to research questions were negative. However, the suspicions that they are affirmative are high. It is important to keep repeating the experiments in order to draw definitive conclusions. We suggest repeating these same experiments in the Symbolic Logic course in order to accumulate more data.

## References


[1] E. W. Dijkstra and S. Scholten, *Predicate calculus and program semantics.* Texts and Monographs in Computer Science, Springer-Verlag, New York, 1990.

[2] D. Gries and F. Schneider, *A logical approach to discrete math. Springer.* Springer, New York, 1993.

[3] C. Homepage, *http://nefud.ldc.usb.ve/CalcLogic*, last accessed 2025/01/25.

[4] F. Flaviani, "Interactive theorem prover based on calculational logic to assist finite difference and summation learning," *In: Uskov, V. L., Howlett, R. J. (eds.) Smart Education and e-Learning 2021. Smart Innovation, Systems and Technologies*, vol. 240, no. -, pp. 161–172, 2021.

[5] Federico Flaviani and Walter Carballosa, "Proof assistant based on calcultional logic to assist the learning of propositional logic and boolean algebras," in *In: XLVIII Latin American Computer Conference (CLEI) 2022*, October 2022, pp. 1–9. Armenia, Colombia.

[6] F. Flaviani and W. Carballosa, "Education-oriented proof assistant based on calculational logic: Proof theory algorithms and assessment Experience," *CLEI Electronic Journal*, vol. 26, no. 2, pp. 1–28, 2023.

[7] F. Flaviani, S. Carrasquel and D. Coronado, "Interactive theorem assistant for learning calculational logic in the style of "a logical approach to discrete math"," in *In: LI Latin American Computer Conference (CLEI) 2025*, October 2025, pp. Valparaiso, Chile.

[8] A. Church, "A set of postulates for the foundation of logic," *Annals of mathematics*, vol. 33, no. 2, pp. 346–666, 1932.

[9] C. H and F. R., *Combinatory logic, vol 1.* North-Holland Publishing Company, Amsterdam, 1958.

[10] E. W. Dijkstra, "The everywhere operator once more," Austin University, TX, Tech. Rep. EWD1086, nov 1990.

[11] D. Gries and F. Scheneider, "Formalizations of substitution of equals for equals," Cornel University, NY, Tech. Rep. TR98-1686, may 1998.

[12] P. C. Y. Bertot, *Interactive theorem proving and program development: Coq Art: The calculus of inductive constructions.* Berlin, Heidelberg: Springer-Verlag, 2004.

[13] M. W. T. Nipkow, L. Paulson, *http://isabelle.in.tum.de/dist/Isabelle2015/doc/tutorial.pdf*, 2015.

[14] L. De Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *International Conference on Automated Deduction.* Springer, 2015, pp. 378–388.

[15] W. A. Howard *et al.*, "The formulae-as-types notion of construction," *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, vol. 44, pp. 479–490, 1980.

[16] P. Martin-Löf, "An intuitionistic theory of types: Predicative part," in *Studies in Logic and the Foundations of Mathematics.* Elsevier, 1975, vol. 80, pp. 73–118.

[17] L. Bijlsma and R. Nederpelt, "Dijkstra-Scholten predicate calculus: Concepts and misconceptions," *Acta Informatica*, vol. 35, no. 1, pp. 1007–1036, 1998.

[18] D. Gries and F. Scheneider, "Equational propositional logic," *Information Processing Letters*, vol. 53, no. 3, pp. 145–152, 1995.

[19] G. Tourlakis, "A basic formal equational predicate logic - Part I," *Bulletin of the Section of Logic*, vol. 29, no. 1, pp. 43–56, 2000.

[20] ——, "A basic formal equational predicate logic - Part II," *Bulletin of the Section of Logic*, vol. 29, no. 3, pp. 75–87, 2000.

[21] ——, "On the soundness and Completeness of Equational Predicate Logics," *J. of Logic and Computation*, vol. 11, no. 4, pp. 623–653, 2001.

[22] ——, "A new foundation of a complete boolean equational logic," *Bulletin of the Section of Logic*, vol. 38, no. 1, pp. 12–28, 2009.

[23] V. Lifschitz, "On calculational proofs," *Annals of Pure and Applied Logic*, vol. 113, no. 1, pp. 207–224, 2001.

[24] R. Dijkstra, ""Everywhere" in predicate algebra and modal logic," *Information Processing Letters*, vol. 58, no. 5, pp. 237–243, 1996.

[25] D. Gries, "Foundations for calculational logic," *Mathematical Methods in Program Development, NATO ASI Series F: Computer and System Sciences. Springer, Berlin*, vol. 158, no. 1, pp. 83–126, 1997.

[26] D. Gries and F. Scheneider, "Adding the everywhere operator to propositional logic," *J. Logic and Comp*, vol. 8, no. 1, pp. 119–130, 1998.

[27] K. Broda, *(2009) slides of automated reasoning course. Imperial college, london. [Online]. Available:*, https://www.doc.ic.ac.uk/ kb/MACTHINGS/SLIDES/0LIntro4up.pdf.

[28] A. Mercer, A. Bundy, H. Duncan, D. Aspinall, "Pg tips: A recommender system for an interactive theorem prover," in *In: Mathematical User-Interfaces Workshop (MathUI 2006)*, Aug. 2006, pp. 290–294. Oxford, London.

[29] W. Kahl, "The teaching tool calccheck a proof-checker for gries and schneider's "logical approach to discrete math"," in *International Conference on Certified Programs and Proofs*. Springer, 2011, pp. 216–230.

[30] ——, "Calccheck: a proof checker for teaching the "logical approach to discrete math"," in *International Conference on Interactive Theorem Proving*. Springer, 2018, pp. 324–341.

[31] A. Mendes and J. F. Ferreira, "Towards verified handwritten calculational proofs," *in: Avigad J., Mahboubi A. (eds) Interactive Theorem Proving. Proc. 9th ITP Oxford. Lecture Notes in Computer Science*, vol. 10895, pp. 432–440, 2018.

[32] S. Böhne and C. Kreitz, "Learning how to prove: From the coq proof assistant to textbook style," in *P. Quaresma and W. Neuper (Eds.): 6th International Workshop on Theorem proving components for Educational software (ThEdu 17) EPTCS 267*, Gothenburg, Sweden, Aug. 2017, pp. 1–18.

[33] H. Simmons, *Derivation and Computation: taking the Curry-Howard correspondence seriously*. Cambridge University Press, 2000.

[34] H. Curry, "The universal quantifier in combinatory logic," *Annals of mathematics*, vol. 32, no. 1, pp. 154–180, 1931.

[35] G. Frege, *Begriffsschrift, 1879. Translated in: From Frege to Godel, J. van Heijenoort (ed.)*. Harvard University Press, 1967.

[36] J. A. Bohórquez, "Intuitionistic Logic according to Dijkstra's Calculus of Equational Deduction," *Notre Dame Journal of Formal Logic*, vol. 49, no. 4, pp. 361–384, 2008.

[37] S. Broda and L. Damas, "Compact Bracket Abstraction in Combinatory Logic," *The Journal of Symbolic Logic*, vol. 62, no. 1, pp. 729–740, 1997.

[38] F. Flaviani and J. E. Tahhan, "Criteria for Bracket Abstraction Design," *Electronic Notes in Theoretical Computer Science*, vol. 349, no. 1, pp. 25–48, 2020.

[39] M. Bunder, "Expedited broda-damas bracket abstraction," *The Journal of Symbolic Logic*, vol. 64, no. 4, pp. 1850–1857, 2000.

[40] T. Johnsson, *Lambda lifting: transforming programs to recursive equations.* In Conference on Functional Programming Languages and Computer Architecture, Nancy. Jouannaud (editor). LNCS 201. Springer Verlag, 1985.

[41] B. Russell and A. N. Whitehead, *Principia Mathematica, Volume 1.* Cambridge at the University Press, 1925.

[42] D. Gries, "A calculational proof of andrew's challenge," Cornell University. Ithaca, New York, Ithaca, New York, Tech. Rep. 96-102, - 1996.

[43] T. Tao, "Machine-assisted proof," *Notices of the American Mathematical Society*, vol. 72, no. 1, pp. 6–13, January 2025.